

Docker for Enterprise Operations Exercises

Some of these exercises ask you to run things on your local machine. In order to do so, you will need Docker as well as some other bash utilities installed; in case this is a problem, the 'infra' node has everything you'll need all pre-configured. **When the instructions refer to 'your' machine, this means either your laptop or the 'infra' node - whichever you have chosen to use.**

Be aware that copy-pasting of commands or code snippets from this PDF may apply changes to some characters e.g. quotes, tabs, which may leads to errors. Please consider typing suggested commands and code snippets in case you encounter any issues.

Windows Users: Please note that in all exercises we will use Unix style paths using forward slashes ('/') instead of backslashes ('\'). On Windows you can work directly with such paths by either using a **Bash** terminal or a **Powershell** terminal. Powershell can work with both Windows and Unix style paths.

Contents

Exercises	5
1 Install UCP	5
1.1 Checking ntp	5
1.2 Installing UCP	5
1.3 Licensing the installation	6
1.4 Checking SANs	6
1.5 Adding additional worker nodes	6
1.6 Configuring Worker Orchestrators	7
1.7 Conclusion	7
2 (Optional) Configuring UCP for High Availability	7
2.1 Checking ntp	7
2.2 Adding manager nodes	8
2.3 Load Balancing UCP	8
2.4 Configuring the Scheduler	9
2.5 Conclusion	10
3 UCP API & Client Bundles	10
3.1 Establishing an Auth Token	10
3.2 Using Client Bundles	11
3.3 Using the API Docs (Optional)	12
3.4 Making a Secret	12
3.5 Creating a Service	13
3.6 Conclusion	13
4 Access Control in UCP	13
4.1 Designing Your RBAC Model	14
4.1.1 Planning Teams & Organizations	14
4.1.2 Establishing Permissions	14
4.1.3 Defining Kubernetes RBAC Rules	14
4.2 Optional: Client Bundles and RBAC	16

4.3	Conclusion	16
5	User Management with LDAP	16
5.1	Setting Up the LDAP Server	16
5.2	Integrate UCP and LDAP	17
5.3	Testing User Access	18
5.4	Populating Teams from LDAP Groups	18
5.5	Populating Teams from LDAP Searches	19
5.6	Cleanup	19
5.7	Conclusion	19
6	Password Recovery	19
6.1	Recovering admin passwords	19
6.2	Conclusion	20
7	Orchestrating Applications	20
7.1	Background info	20
7.2	Deploying to Swarm	21
7.3	Scaling the application	22
7.4	Deploying to Kubernetes	23
7.5	Deploying Application Packages	24
7.6	Conclusion	25
8	Combining Collections and Kubernetes Namespaces	25
8.1	Creating Collections	25
8.2	Associating Worker Nodes with Collections	25
8.3	Creating Kubernetes Namespaces	25
8.4	Associating Nodes with Namespaces	26
8.5	Creating a Deployment in the development namespace	26
8.6	Creating a Deployment in the production namespace	27
8.7	Cleaning up	27
8.8	Conclusion	27
9	Basic Swarm Routing Models	27
9.1	Routing Cluster-Internal Traffic	28
9.1.1	Routing to Stateless Services	28
9.1.2	Routing to Stateful Services	29
9.2	Routing Ingress Traffic	29
9.2.1	Ingress Traffic to Stateless Services	29
9.2.2	Ingress Traffic to Stateful Services	30
9.3	Conclusion	31
10	Basic Kubernetes Routing Models	31
10.1	Routing Cluster-Internal Traffic	31
10.1.1	Routing to Stateless Deployments	31
10.1.2	Routing to Stateful Deployments	33
10.2	Routing Ingress Traffic	34
10.2.1	Ingress Traffic to Stateless deployments	34
10.3	Conclusion	34
11	L7 Swarm Routing with Interlock	35
11.1	Enabling Interlock	35
11.2	Deploying Services with Interlock	35
11.3	Configuring Sticky Sessions	37
11.4	Conclusion	37
12	Kubernetes Ingresses	37
12.1	Setting up an IngressController in UCP	38

12.2	Configuring L7 and Path-Based Routing	39
12.3	Configuring Sticky Sessions	39
12.4	Conclusion	41
13	Release Models in Swarm	41
13.1	Blue / Green Releases	41
13.2	Canary Releases	42
13.3	Conclusion	43
14	Release Models in Kubernetes	43
14.1	Blue / Green Releases	43
14.2	Canary Releases	45
14.3	Conclusion	46
15	Configuring Engine Logs	46
15.1	Setting the Logging Driver	46
15.2	Configuring Log Compression and Rotation	47
15.3	Conclusion	48
16	UCP Audit Logs	48
16.1	Configuring Audit Logs	48
16.2	Conclusion	49
17	Centralized Logging	49
17.1	Installing ELK Stack	49
17.2	Configuring all Swarm nodes	50
17.3	Stream all Docker logs to ELK	50
17.4	Conclusion	52
18	Health Checks	52
18.1	Analyzing the Dockerfile	52
18.2	Deploying a Healthcheck-Enabled Service	53
18.3	Running a Pod with a Liveness Probe	54
18.4	Conclusion	55
19	Installing Docker Trusted Registry	55
19.1	Add additional worker nodes to UCP	55
19.2	Installing DTR	55
19.3	Conclusion	56
20	(Optional) Configuring DTR for High Availability	56
20.1	Setting up a DTR Storage Backend	56
20.2	Installing Replicas	57
20.3	Load Balancing DTR	58
20.4	Cleanup	58
20.5	Conclusion	59
21	Pushing and Pulling From DTR	59
21.1	Integrate UCP and DTR	59
21.2	Pushing your first image	60
21.3	Conclusion	60
22	Working with Organizations and Teams	61
22.1	Creating Organizations and Teams	61
22.2	R/W in Org Repos	61
22.3	Establishing Access Tokens	62
22.4	Conclusion	62

23 Content Trust	62
23.1 Setup	62
23.2 Establishing Trust	62
23.3 Granting Signing Authority	65
23.4 Requiring Trust at Runtime	66
23.5 Revoking Trust Metadata	67
23.6 Revoking Signing Authority	67
23.7 Optional: Enforcing Trust Pinning	68
23.8 Conclusion	69
24 Image Promotion & Webhooks	69
24.1 Creating a Promotion Pipeline	69
24.2 Setting up a Webhook	69
24.3 Triggering the Pipeline	70
24.4 Auditing DTR Events	70
24.5 Optional: Image Mirroring	70
24.6 Conclusion	71
25 Tag Pruning and Garbage Collection	71
25.1 Pruning Tags	71
25.2 Configuring Pruning Policies	72
25.3 Configuring Garbage Collection	72
25.4 Conclusion	72
26 (Optional) Content Caching	72
26.1 Setup	73
26.2 Establishing a Content Cache	73
26.3 Pulling from a Content Cache	75
26.4 Conclusion	76
27 The Software Supply Chain	76
27.1 Your Pipeline	76
27.2 Tasks	76
27.3 Optional Challenges	77
27.4 Conclusion	77
28 Appendix: Build Server	77
28.1 Prerequisites	78
28.2 Creating and Shipping the Jenkins Image	78
28.3 Preparing the Repository in DTR	79
28.4 Preparing a Source Repo	79
28.5 Running Jenkins in the Swarm	79
28.5.1 Preparing the Swarm Node	79
28.5.2 Creating the Jenkins Service	79
28.5.3 Finalizing the Jenkins Configuration	80
28.6 Configuring Jenkins Jobs	80
28.7 Conclusion	81
Instructor Demos	81
1 Instructor Demo: Containerized Nature of UCP	81
1.1 Containers on UCP Manager and Worker Nodes	81
1.2 Containerized Nature of Kubernetes	83
1.3 Supporting Resources	83
1.4 Conclusion	84
2 Instructor Demo: UCP RBAC	84
2.1 Part 1: Basic Swarm RBAC	85

2.2	Part 2: Resource Collections, Teams, and Organizations	85
2.3	Part 3: Kubernetes Namespaces & Role Bindings	85
2.4	Conclusion	86
3	Instructor Demo: Security Scanning	86
3.1	Setting Up Security Scanning	86
3.2	Configuring Scanning Options	87
3.3	Scanning an Image	87
3.4	Reading Scan Results	88
3.5	Overriding a Vulnerability	89
3.6	Vulnerabilities in UCP	90
3.7	Conclusion	90

Exercises

1 Install UCP

By the end of this exercise, you should be able to:

- Install Universal Control Plane on a single manager node
- Join worker nodes to UCP
- Confirm correct basic configuration of UCP, including licensing, certificates, and clock sync.

Pre-requisites:

- Three nodes running centos:7.5 or later, and Docker EE 18.09-ee or later, named as per:
- ucp-manager-0
- ucp-node-0
- ucp-node-1
- A UCP license. If you don't have one, a trial license is available at <https://store.docker.com/bundles/docker-datacenter/purchase?plan=free-trial>.

1.1 Checking ntp

1. While not strictly required, the `ntp` utility ensures clock synchronization across your cluster, helping you avoid a number of related problems. Check that `ntp` is running on `ucp-manager-0`, `ucp-node-0` and `ucp-node-1`:

```
[centos@ucp-manager-0 ~]$ sudo /bin/systemctl status ntpd.service
ntpd.service - Network Time Service
   Loaded: loaded (/usr/lib/systemd/system/ntpd.service; enabled; vendor preset: disabled)
   Active: active (running) since Sat 2017-10-14 02:14:40 UTC; 39min ago
   ...
```

Note it's important that `ntp` is up and running *before* installing UCP.

1.2 Installing UCP

1. On `ucp-manager-0`, use the UCP bootstrapper to install UCP:

```
[centos@ucp-manager-0 ~]$ UCP_IP=<ucp-manager-0 public IP>
[centos@ucp-manager-0 ~]$ UCP_FQDN=<ucp-manager-0 FQDN>
[centos@ucp-manager-0 ~]$ docker container run --rm -it --name ucp \
-v /var/run/docker.sock:/var/run/docker.sock \
docker/ucp:3.1.1 install \
```

```
--admin-username admin \
--admin-password adminadmin \
--san ${UCP_IP} \
--san ${UCP_FQDN}
```

1.3 Licensing the installation

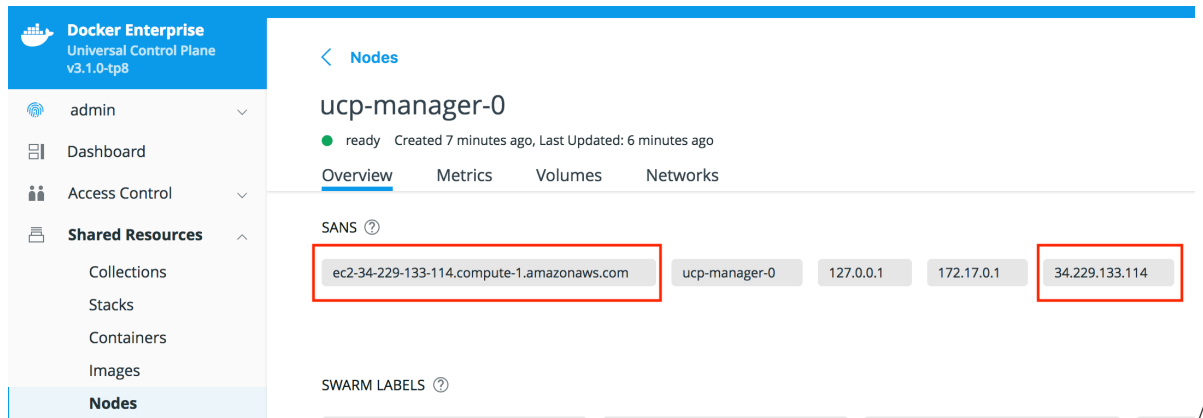
1. In your browser, visit the UCP interface at `https://<public IP of ucp-manager-0>`.

Note: Most browsers will display some sort of warning message regarding the connection. This is because we are using the default self signed certificates as opposed to a certificate from a trusted source. You can tell the browser to ignore the warning and proceed.

2. Login to your Admin account; as per the flags in the install command above, your username is `admin` and your password is `adminadmin`.
3. Once logged in you will see a prompt screen asking you to upload a license. Click the **Upload License** button and upload your UCP license; if you need a trial license, click the **Get free trial license** button, and follow the prompts.

1.4 Checking SANs

1. On the left sidebar, navigate **Shared Resources** -> **Nodes** -> **ucp-manager-0**, and scroll down in the resulting view to where it says **SANS**. Confirm that the FQDN and public IP of the `ucp-manager-0` node are in the list. You specified these at install via the `--san` flag to the UCP bootstrapper. **Failing to specify SANs correctly will cause TLS failures when communicating with other nodes if left unaddressed**, since these SANs are added to this node's certificate.



1.5 Adding additional worker nodes

1. Navigate **Shared Resources** -> **Nodes** -> **Add Node**. A swarm join token is presented which can be used to add workers to your UCP cluster.
2. Cut and paste the `docker swarm join...` token presented in the last step in the terminal on `ucp-node-0` and `ucp-node-1`. After a moment, they should be visible on the **Nodes** view of UCP as workers.

STATUS	NAME	TYPE	ROLE	ADDRESS	ENGINE	OS/ARCH	CPU	MEMORY	DISK	DETAILS
●	ucp-node-0	swarm	worker	10.10.0.177	18.09.0-beta5	linux/x86...	3.45%	4.59%	17.42%	Healthy UCP worker
●	ucp-node-1	swarm	worker	10.10.25.142	18.09.0-beta5	linux/x86...	1.3%	4.55%	17.44%	Healthy UCP worker
●	ucp-manager-0	mixed	manager	10.10.40.29	18.09.0-beta5	linux/x86...	17.86%	13.68%	19.94%	Healthy UCP manager

1.6 Configuring Worker Orchestrators

By default, the two workers you just added will only accept Swarm workloads; let's configure them to accept Swarm or Kubernetes workloads.

Note: Mixed orchestrator configuration is not recommended in production. For critical clusters, allocate some nodes for Swarm workloads, and some nodes for Kubernetes workloads, as needed.

1. Navigate **Shared Resources** -> **Nodes** -> **ucp-node-0**. Click on the gear in the top right to edit this node's configuration.
2. Under *Orchestrator Type*, click **Mixed**.
3. Click **Save** in the bottom right.
4. Repeat these steps for ucp-node-1.

1.7 Conclusion

At this point, UCP is installed with one manager and two worker nodes. Bear in mind that UCP is really just a collection of containers running on a swarm. Alternatively, we could have set up a swarm from the command line first, and installed UCP on top of this if we preferred; UCP would have automatically recognized and integrated all nodes.

2 (Optional) Configuring UCP for High Availability

By the end of this exercise, you should be able to:

- Add managers to a UCP management consensus to provide high availability for UCP
- Configure a load balancer to distribute traffic across a management consensus
- Prevent workload from getting scheduled on any UCP manager or DTR replica

Pre-requisites:

- UCP installed with 2 worker nodes
- Two additional nodes running centos:7.5 or later, and Docker EE 18.09-ee or later, named:
- ucp-manager-1
- ucp-manager-2

2.1 Checking ntp

1. Like we did for our other UCP nodes, make sure ntp is running on ucp-manager-1 and ucp-manager-2:

```
[centos@ucp-manager-1 ~]$ sudo /bin/systemctl status ntpd.service
[centos@ucp-manager-2 ~]$ sudo /bin/systemctl status ntpd.service
```

If it isn't, `sudo /bin/systemctl start ntpd.service` should start the service.

2.2 Adding manager nodes

1. Return to the same UCP page visited previously to get a join token for workers.
2. Move the toggle from **Worker** to **Manager**. Note that the join token changes.
3. Open terminal connections to `ucp-manager-1` and `ucp-manager-2`, and use this new join command to add these nodes as managers.
4. Go back to the **Nodes** page on UCP. You should now see 5 nodes, similar to the following:

STATUS	NAME	TYPE	ROLE	ADDRESS	ENGINE	OS/ARCH	CPU	MEMORY	DISK	DETAILS
●	ucp-node-0	swarm	worker	10.10.0.177	18.09.0-beta5	linux/x8...	3.97%	4.92%	17.42%	Healthy UCP worker
●	ucp-node-1	swarm	worker	10.10.25.142	18.09.0-beta5	linux/x8...	5.13%	4.94%	17.42%	Healthy UCP worker
●	ucp-manager-1	mixed	manager	10.10.51.124	18.09.0-beta5	linux/x8...	8.27%	10.84%	19.94%	Healthy UCP manager
●	ucp-manager-2	mixed	manager	10.10.68.0	18.09.0-beta5	linux/x8...	57.5%	10.26%	19.95%	Healthy UCP manager
●	ucp-manager-0	mixed	manager	10.10.40.29	18.09.0-beta5	linux/x8...	13.85%	15.47%	19.97%	Healthy UCP manager

Note: It takes a few minutes for a node to be set up as a UCP manager. This is reflected in the **Details** column. At first it will say that the node is *Pending*, and when finished it will say *Healthy UCP Manager*. Red warning banners at the top of UCP are normal during node configuration, and should resolve themselves once the nodes finish setting up. Be patient! Interrupting the manager join process is a common way to corrupt a cluster.

2.3 Load Balancing UCP

Now that we have three UCP managers, we'd like to load balance requests to the UCP API and frontend across them. We'll configure an nginx load balancer on your `infra` node for this purpose.

1. First we have to reconfigure UCP's certificates to accept traffic from our load balancer. On UCP, navigate **Shared Resources** -> **Nodes** -> **ucp-manager-0**, then click on the gear icon that appears in the top right to edit this node's configuration.
2. Scroll down to **SANs**, click **Add SAN**, and place the public IP of your `infra` node in the box that appears; do the same with the FQDN for the `infra` node. Click **Save** in the bottom right. This configures UCP's certs on this node to accept requests from the `infra` node.
3. Refresh your browser a few times over the course of the next minute. You may see some warnings or errors from `ucp-manager-0`; this is normal while certificates are being updated. Once you refresh and your browser asks you to accept a new certificate, the certificate update process for this node is complete.
4. Repeat steps 1-3 for `ucp-manager-1` and `ucp-manager-2`.
5. On your `infra` node, create a file called `ucp-nginx.conf`, and place the following content in it; make sure to replace the `<variables>` with the public IPs of your own manager nodes:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
```



```

    worker_connections 1024;
}

stream {
    upstream ucp_443 {
        server <ucp-manager-0 public IP>:443;
        server <ucp-manager-1 public IP>:443;
        server <ucp-manager-2 public IP>:443;
    }
    upstream ucp_6443 {
        server <ucp-manager-0 public IP>:6443;
        server <ucp-manager-1 public IP>:6443;
        server <ucp-manager-2 public IP>:6443;
    }

    server {
        listen 443;
        proxy_pass ucp_443;
    }
    server {
        listen 6443;
        proxy_pass ucp_6443;
    }
}

```

The 443 block is for the UCP API, and the 6443 block is so kubectl can reach the Kube API directly.

6. Deploy a containerized nginx server, mounting the config you just created:

```

[centos@infra ~]$ docker container run -d --name ucp-lb --restart=unless-stopped \
--publish 443:443 \
--publish 6443:6443 \
--volume $(pwd)/ucp-nginx.conf:/etc/nginx/nginx.conf:ro \
nginx:stable-alpine

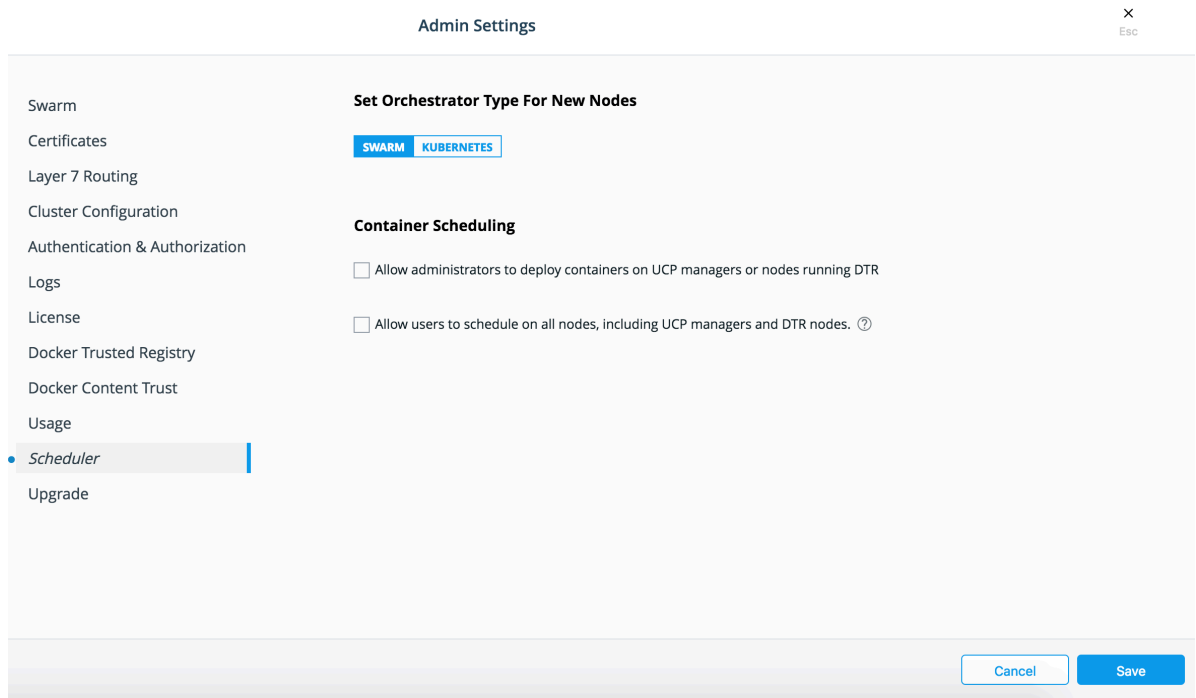
```

7. Visit UCP at <https://<infra node public IP>>. You may be asked to accept several certificates; accept them all, and nginx will forward your traffic onto the UCP management consensus via HTTPS-passthrough round-robin load balancing.

2.4 Configuring the Scheduler

Now that we have 3 manager nodes setup, we want to make sure that our manager nodes are dedicated for their purpose and that they do not run any application containers.

1. In UCP, navigate **admin -> Admin Settings -> Scheduler**.
2. Uncheck both of the checkbox settings on the page as shown below, and be sure to click **Save** afterwards:



This will prevent containers from being scheduled on any of our UCP manager nodes.

2.5 Conclusion

Your UCP cluster now has three manager nodes, preventing your cluster from collapsing if one of them fails. Bear in mind that this manager consensus is exactly a Swarm manager consensus; UCP controllers are just Swarm managers, with additional containers running on top to serve UCP. We have also made our cluster more stable by preventing workload from being scheduled on UCP managers or DTR replicas; this way, if any poorly engineered or managed containers take down a node, it won't be one of the managers controlling your cluster. Finally, note that when load balancing a UCP manager consensus, it is *mandatory* to passthrough HTTPS; do not terminate at the load balancer.

3 UCP API & Client Bundles

By the end of this exercise, you should be able to:

- Authenticate with the UCP API through either a password or an authorization token
- Download and set up a UCP client bundle in order to securely issue commands to a remote Docker Engine from a local Docker client
- Construct and issue calls to the UCP API

Pre-requisites:

- UCP installed and running.

In this exercise, you'll manipulate your Universal Control Plane from a remote machine. Try these exercises on your own laptop if you're able to install Docker there; if not, try them from your `infra` node.

3.1 Establishing an Auth Token

1. In order to authenticate with UCP, we need an auth token. Fetch and save one to `$AUTHTOKEN` with the following commands; note that `UCP_FQDN` will be the FQDN of your `infra` node if you set a load balancer up there for UCP, or the FQDN of `ucp-manager-0` directly if you're not using a load balancer.

```
[centos@infra ~]$ UCP_FQDN=<UCP FQDN>
[centos@infra ~]$ AUTHTOKEN=$(curl -sk \
  -d '{"username":"admin","password":"adminadmin"}' \
  https://${UCP_FQDN}/auth/login | jq -r .auth_token)
```

Note you could do this as any user, not just admin; your ability to take action through the API is limited by the same RBAC rules as the web client.

2. For convenience, create an alias for issuing API commands:

```
[centos@infra ~]$ alias ucp-api='curl -k -H "Authorization: Bearer $AUTHTOKEN"'
```

3.2 Using Client Bundles

This section requires Docker, kubectl, a bash shell, curl and jq to be installed on your local machine. All of this is pre-configured on your infra node if you don't want to run it locally.

One way to control UCP is to issue Docker CLI commands from a remote node to a UCP cluster. Those commands will be governed by the same access control rules imposed on the authenticated user as when they interact with UCP from the web UI or API.

1. Use the UCP API to download the client bundle for your UCP account:

```
[centos@infra ~]$ ucp-api https://${UCP_FQDN}/api/clientbundle -o bundle.zip
```

2. Point your local Docker CLI at your remote UCP using the certificates in the bundle.zip you just downloaded:

```
[centos@infra ~]$ unzip bundle.zip
[centos@infra ~]$ eval "$(<env.sh)"
```

3. Issue any Docker CLI commands you like; the response will reflect your UCP cluster, rather than your local machine:

```
[centos@infra ~]$ docker container ls
[centos@infra ~]$ docker volume create bundle-vol
```

Check that that volume got created via your UCP web UI. Which node did it get created on, and why?

4. These same credentials will also allow you to issue kubectl commands to the Kubernetes assets in your cluster. For example, put the following in a file pod.yml on your infra node:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

Then create this as a pod, and confirm it exists on your UCP web UI:

```
[centos@infra ~]$ kubectl create -f pod.yml
```

5. You can also use the client bundle credentials to hit the Kubernetes master API directly:

```
[centos@infra ~]$ curl -s --cacert ./ca.pem --key ./key.pem --cert ./cert.pem \
  https://${UCP_FQDN}:6443/api/v1/pods
```

You should see all the system pods UCP spins up to provide Kubernetes functionality.

6. Delete your test pod, and point your local CLI back at your local Docker engine by unsetting all the environment variables defined in `env.sh`:

```
[centos@infra ~]$ kubectl delete pod nginx
[centos@infra ~]$ unset DOCKER_TLS_VERIFY COMPOSE_TLS_VERSION \
  DOCKER_CERT_PATH DOCKER_HOST
```

3.3 Using the API Docs (Optional)

Besides the CLI, authenticated commands can be issued to UCP by posting JSON data to UCP's RESTful API.

1. Make sure a couple of non-system containers are running on your Swarm (anything will do). For example, in UCP, navigate **Swarm** -> **Services** -> **Create**, name your service `api-demo`, and specify `nginx:latest` as the image, and click **Create**.
2. From UCP's left sidebar, navigate **Live API** -> **GET** under the 'Container' heading. This provides usage examples on this command.
3. Click **Try It Out** -> **Execute** without filling in any of the optional fields. The API call is constructed and sent, showing an example of the syntax as a `curl` command, and the resulting output.
4. Back on your local machine where you set up your auth token, try:

```
[centos@infra ~]$ ucp-api https://{UCP_FQDN}/containers/json | jq
```

The same result as the demo is produced (`jq` here pretty-prints the json returned by the curl. Feel free to substitute any other utility that does so).

3.4 Making a Secret

1. Create a file `mysecret.dat` on your `infra` node containing any short piece of text you'd like to distribute as a secret across your swarm.
2. The syntax to create a secret is described in the API docs under the POST request for the route `secrets/create`. But, the secret content must be base 64 encoded. Register your secret in your cluster with the correct encoding:

```
[centos@infra ~]$ ucp-api \
  --data '{"Data":"'$(base64 mysecret.dat)'',"Name":"test_secret"}' \
  -X POST https://{UCP_FQDN}/secrets/create
```

3. From the web interface, make a service and mount the secret:
 - From the left sidebar: **Swarm** -> **Services** -> **Create**.
 - Name it `demo`, and specify as an image `"nginx"`.
 - Attach the secret: in the service configuration navigate **Environment** -> **Use Secret** +; select `test_secret` from the *Secret Name* dropdown.
 - Hit **Confirm**, then **Create**.
4. Now, we want to check that the secret we made via the API got mounted:
 - Navigate **Swarm** -> **Services** -> **demo** -> **Metrics**, and scroll all the way to the bottom; there is a list of container for this service there.
 - Click on the one container running for the `demo` service, then click on the `**>_` icon in the top-right.
 - Enter `sh` in the text box that appears, and click **Run** to get a shell connection to this container.
 - Run `cat /run/secrets/test_secret` in the shell, and make sure you get back the secret value you created via the API.

3.5 Creating a Service

1. Use the API to inspect the demo service you created in the last step (hint: start back at the API docs page to choose an appropriate endpoint). Notice that even this simple service has a large JSON object describing it.
2. Use the API to delete your demo service. Recall `curl` makes a DELETE request when the flag `-X DELETE` is used.
3. A similar JSON object to the one found above is used to create a new service. Entering this at the command line would be a nuisance, so create a basic service definition in a file `myservice.json` on your `infra` node:

```
{
  "Name": "demo",
  "TaskTemplate": {
    "ContainerSpec": {
      "Image": "nginx:latest"
    }
  }
}
```

4. Create a service using the definition in `myservice.json`:

```
[centos@infra ~]$ ucp-api --data @myservice.json https://{UCP_FQDN}/services/create
```

Check to see that your service is running in the web UI as you'd expect, and delete it.

5. Modify the content of `myservice.json` so that port 80 internal to the container is exposed on port 8080 external to the container.

Hint 1: check the 'Try It Out' example in the API docs for the POST to `/service/create` for an example of the JSON structure you'll need.

Hint 2: explore the Docker API docs <https://docs.docker.com/engine/api/v1.30/> for detailed explanations of how most Docker client commands map to API endpoints.

Re-create the service as above, and make sure you can see the nginx landing page at `<ucp-node-0 public IP>:8080`.

6. Finally, modify `myservice.json` again to attach the `test_secret` secret to this service. Launch the service and check that the secret was mounted correctly.

3.6 Conclusion

Anything that can be done in the UCP UI can be done through the API illustrated in this exercise. Under the hood, the UCP web app is issuing calls to the exact same API, all reachable on port 443 of any UCP manager node. All the same holds true for DTR's web interface and underlying API. Also note, there's nothing special about `curl` in this context; any tool to issue these requests, like Chrome's postman, will have the same effect.

4 Access Control in UCP

In this exercise, we'll describe a simple set of users and assets at a startup called Whalecorp, and create the entities and permissions described in your UCP instance.

Pre-requisites:

- A working UCP installation with at least two worker nodes, `ucp-node-0` and `ucp-node-1`

4.1 Designing Your RBAC Model

4.1.1 Planning Teams & Organizations

For the following steps, sketch out teams and organizations with *pen and paper*. We'll click through UCP later, but first we need to come up with an appropriate plan.

1. Whalecorp has the following members:

- **Development:** Joey and Shaun
- **QA:** Kelly and Barry
- **Operations:** Chloe

You anticipate that these groups of employees will need different permissions to different groups of resources. Design a UCP organization and team structure for Whalecorp that will accommodate this.

2. Once your sketch is complete, create the corresponding users, teams and organizations in UCP. Remember from the demo, that all these objects are found under **Access Control** in UCP's right sidebar.

4.1.2 Establishing Permissions

Now, we'd like to grant our organization and teams permissions to interact with resource collections.

1. The app that Whalecorp is currently developing has the following characteristics:

- It will be orchestrated on Swarm.
- It requires an overlay network and a secret.
- Development and QA staff will each run their own separate instance, and should not be able to see or affect the other's.
- Operations staff should be able to access and manipulate both instances with *a single grant*.

Get your pen & paper again and sketch some resource collections for these objects, and choose some roles to associate these collections to the teams you defined previously that will enable the access described above.

2. Create the collections you just planned in UCP under **Shared Resources->Collections->View Children**.
3. Create the corresponding UCP grants under **Access Control->Grants->Swarm**.
4. In order to avoid collisions between Development and QA, update your resource collections such that Joey and Shaun can only deploy services to `ucp-node-0`, and Kelly and Barry can only deploy services to `ucp-node-1`. Chloe can deploy on either machine.
5. During development, developers (Joey and Shaun) may need to exec into a running container to examine their application in-flight. But, the QA employees shouldn't be able to exec into a running container, so that secrets aren't exposed to them. Meanwhile, Chloe should have unrestricted access to all running containers. Change the roles you granted above if necessary to ensure that this is the case.
6. Clean up by deleting the networks and secrets you made above (feel free to leave the users, teams and organizations, resource collections and grants alone).

4.1.3 Defining Kubernetes RBAC Rules

In this part, we want to grant the QA staff read-only access to a namespace called `development`.

1. First, we need to create the `development` namespace.
 - In UCP, navigate **Kubernetes -> + Create**.
 - Add the following YAML to the text box, and click **Create**:

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

- The new namespace should appear in the list of namespaces.

2. Create a custom Kubernetes Role:

- Navigate to **Access Control->Roles->Create**.
- Select **development** from the Namespace dropdown.
- Add following YAML to the text box:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-development
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

- Click **Create**.

3. In order to be able to see something happening within the namespace, we will create a pod.

- Navigate **Kubernetes -> + Create**.
- Add the following YAML to the text box and click **Create**:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: development
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

You'll be redirected to the pods list, where the new pod **nginx** should be visible.

Note: You might need to set the context to development: navigate **Kubernetes -> Namespaces**, hover over *development*, and click **Set Context**.

4. In order to grant our QA staff access to the namespace, we need to create a *RoleBinding*:

- Navigate to **Access Control -> Grants -> Kubernetes -> Create Role Binding**.
- Under **Subject**, click **Organizations**, and from the dropdown menus, select the organization you made in the first part of this exercise, and then select the team in this org you set for the QA staff. Click **Next**.
- Under **Resource Set**, click **Select Namespace** to the right of **development**.
- Under **Role**, click **Cluster Role**, then select **read-development** from the dropdown.
- Click **Create**.

You will be redirected back to the overview of Kubernetes grants, where your RoleBinding *<org name>-<team-name>:read-development* should be visible.

5. Now, let's check whether our pod read permissions are set correctly:

- Login to UCP as one of the members of the QA staff (Kelly or Barry).
- Navigate to **Kubernetes -> Namespaces** and set context to the development namespace like above.
- Navigate to **Kubernetes -> Pods**. At this level, you should be able to see the **nginx** pod you created.
- Hover over the pod's table row, click the ... icon on the right, and select **remove**.

A notification appears saying that the pod cannot be deleted due to denied access, which is the expected behavior, since our QA staff has read-only access to the development namespace.

6. Clean up by logging back into UCP as an admin and deleting your pod.

4.2 Optional: Client Bundles and RBAC

1. On your `infra` node, fetch and source a client bundle for a non-admin user (see the instructions in the *UCP API & Client Bundles* exercise if you're not sure how to do this).
2. Create a service via the CLI on `infra`, such as `docker service create --name rbacdemo nginx`.
3. In UCP, navigate to **Swarm** -> **Services** -> **rbacdemo**. What resource collection has this service been placed in, and why?
4. Again from `infra`, place the following in a file `rbacpod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: rbacpod
spec:
  containers:
  - name: demo-nginx
    image: nginx
```

5. Create this pod via `kubectl create -f rbacpod.yaml`. Find the pod in UCP; what namespace has it been placed into, and why?
6. Clean up by logging back into UCP as an admin and deleting the service and pod you created in this section.

4.3 Conclusion

Planning out a role based access control model before touching UCP is a good way to avoid redundancy and difficult-to-manage grant lists. Often, the fewer grants you can make, the better; that way, your list of grants is easy to audit and understand. Leverage the 'trickle down' model of RBAC that UCP uses for Swarm governance, when appropriate: grants to organizations apply to all teams and all users in those organization, and grants to parent resource collections apply to child resource collections. This trickle down structure is useful for locking down low- or no-permissions generally, and for granting general permissions to the team leaders or senior staff who need them. For Kubernetes, remember that permissions can be confined to a namespace via a `RoleBinding`, or granted across namespaces with a `ClusterRoleBinding`; most Kube permissions for most users should be at the namespace, `RoleBinding` level to avoid accidentally elevating privileges across the entire cluster.

5 User Management with LDAP

By the end of this exercise, you should be able to:

- Configure UCP to pull users from an existing LDAP server
- Populate UCP teams via LDAP groups or searches

Pre-requisites:

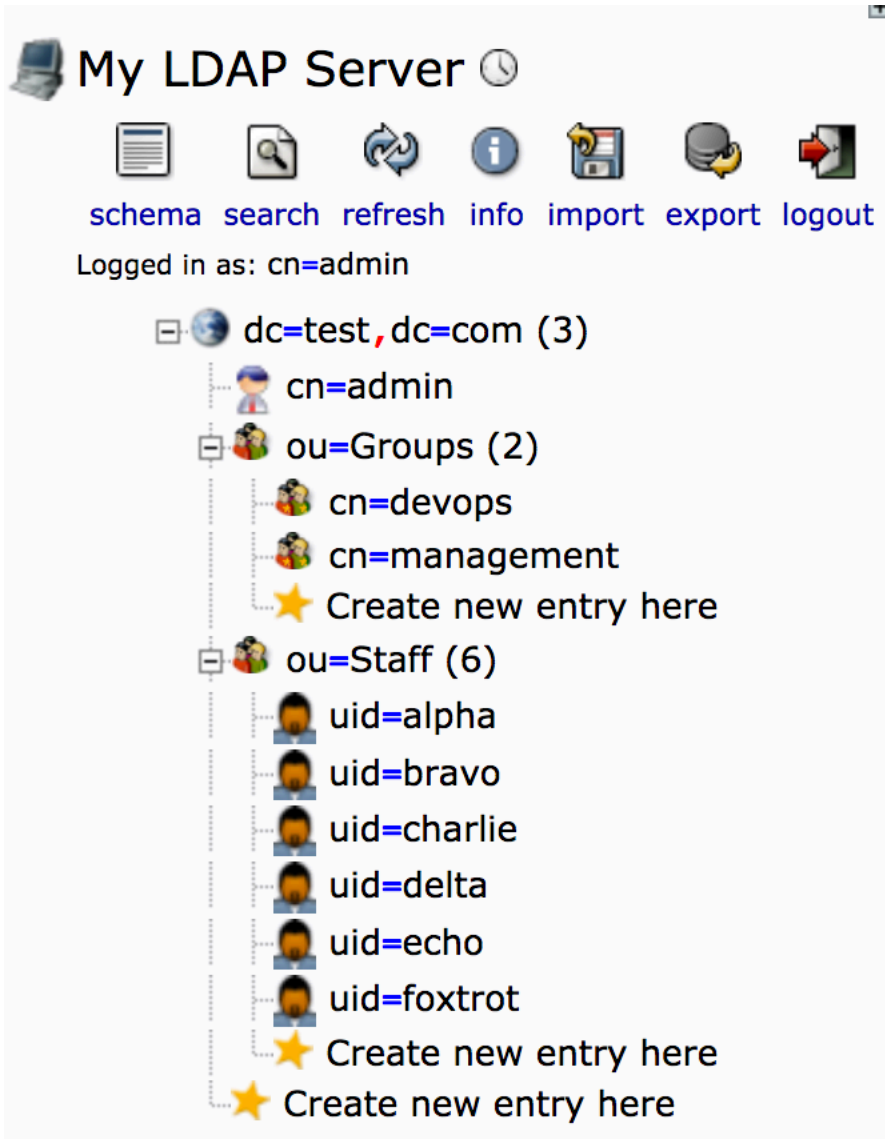
- UCP installation.

5.1 Setting Up the LDAP Server

1. We'll set up a demo LDAP server on your `infra` node. First, make sure the Docker client is pointing at the local Docker engine by doing `echo $DOCKER_HOST` - there should be an empty response. If not, your `infra` node may still be pointing at your UCP cluster from when we were investigating client bundles; unset the environment variables set in `env.sh` for now.
2. On your `infra` node, spin up a containerized LDAP demo:


```
[centos@infra ~]$ docker container run -d -p 80:80 -p 389:389 training/ldap:latest
```

- Open a browser and go to `http://<infra FQDN>/phpldadmin`. This will open up the GUI for your LDAP server.
- Login to the server with the following credentials:
Login DN: `cn=admin,dc=test,dc=com` **Password:** `admin`
- Expand the left navigation bar and take note of the LDAP entries underneath `ou=Staff`, and `ou=Groups`:



5.2 Integrate UCP and LDAP

- Login to UCP as the admin user.
- Navigate **admin** -> **Admin Settings** -> **Authentication and Authorization**, and click **YES** under *LDAP Enabled*.
- In the *LDAP Server* form, fill in:
 - LDAP Server URL: `ldap://<infra FQDN>`
 - Reader DN: `uid=bravo,ou=Staff,dc=test,dc=com`

- Reader Password: password
4. Click **Add LDAP User Search Configurations** + and fill in:
 - Base DN: dc=test,dc=com
 - Username Attribute: uid
 - Fullname Attribute: cn
 - Filter: objectClass=inetOrgPerson
 - Check **Search subtree instead of just one level**
 - Click **Confirm**
 5. Under *LDAP Test Login*, fill in:
 - Username: bravo
 - Password: password
 6. Still under *LDAP Test Login*, click **Test**. A successful login test should be reported.
 7. Under *LDAP Sync Configuration*, enter 24 for the **Sync Interval (Hours)** field.
 8. Click **Save**.

5.3 Testing User Access

1. In the *LDAP Sync Jobs* section, click on the **Sync Now** button.
2. Return to the UCP dashboard, and navigate **Access Control** -> **Users**. None of the LDAP users appear listed here. UCP only moves user accounts over when the user first logs in.
3. Logout of UCP as the admin user, and log back in as user bravo / password password.
4. Log back in as the admin user, and check the users list again. bravo now appears listed here.

5.4 Populating Teams from LDAP Groups

1. Go back to the LDAP configuration page (admin -> **Admin Settings** -> **Authentication & Authorization**):
 - uncheck *Just-in-time User Provisioning*
 - Save the config
 - click **Sync Now**.

This is not strictly necessary for populating teams, but will make it easier to see who is getting put in your teams without having to log in as each one to trigger the JIT provisioning.

2. Create an engineering organization via **Access Control** -> **Orgs & Teams**.
3. Make a team devops in your engineering organization, and click **Yes** under *Enable Sync Team Members* when creating the team.
4. Click **Match Group Members** to populate this team based on an LDAP group.
5. Fill out the subsequent fields:
 - Group DN: cn=devops,ou=groups,dc=test,dc=com
 - Group Member Attribute: uniqueMember
6. Click **Yes** under *Immediately Sync Team Members*, and finally click **Create**.
7. Navigate back to your new devops team users listing; the same users in the devops team in the LDAP browser should now be members of your devops team in UCP.

5.5 Populating Teams from LDAP Searches

1. Make a team staff in your engineering organization, and click **Yes** under **Enable Sync Team Members** when creating the team.
2. Click **Match Search Results** to populate this team based on an LDAP search.
3. Enter `dc=test,dc=com` for *Search Base DN*
4. Enter `ou=Staff` for *Search Filter*
5. Check **Search subtree instead of just one level**, and **Yes** for *Immediately sync team members*
6. Click **Create** at the bottom, and navigate to the user list for the staff team. The team is populated with users matching your search criteria.

5.6 Cleanup

1. Disable the LDAP integration by selecting **No** under *LDAP Enabled* on the **Authentication and Authorization** admin settings page.
2. Click **Save**.

5.7 Conclusion

In this exercise, we imported users from an LDAP server. Users will be populated with just-in-time provisioning on their first login by default, unaffiliated with any team. UCP teams can be populated from LDAP groups or by LDAP searches.

6 Password Recovery

By the end of this exercise, you should be able to:

- Reset the UCP administrator's password from any UCP manager.

Pre-requisites:

- Make sure UCP is not using LDAP for its user accounts (see **Admin Settings -> Authentication and Authorization**).

6.1 Recovering admin passwords

1. SSH into the `ucp-manager-0` node.
2. If we do a quick `docker container ls` we can see all our UCP containers; the container we want to access is `ucp-auth-api`.
3. Run the following commands:

```
[centos@ucp-manager-0 ~]$ authapiid=$(docker container ls \
| grep ucp-auth-api | cut -d' ' -f1)
[centos@ucp-manager-0 ~]$ docker container exec -it ${authapiid} enzi \
"$$(docker container inspect --format '{{ index .Args 0 }}' ${authapiid})" \
passwd --interactive
```

4. You will be prompted for the username. Specify `admin`, then choose a new password. Finally, check that you can log in to the UCP UI with your new credentials.
5. Optional: set your `admin` password back to `adminadmin`. This is not required, but some instructions in future exercises assume this configuration; if you don't want to change your password back, you'll have to remember to provide your new password as needed.

6.2 Conclusion

Note that node access to a UCP controller and the ability to exec-attach as per the above command is all that is required to gain admin control to UCP; make sure access to these nodes is appropriately restricted.

7 Orchestrating Applications

By the end of this exercise, you should be able to:

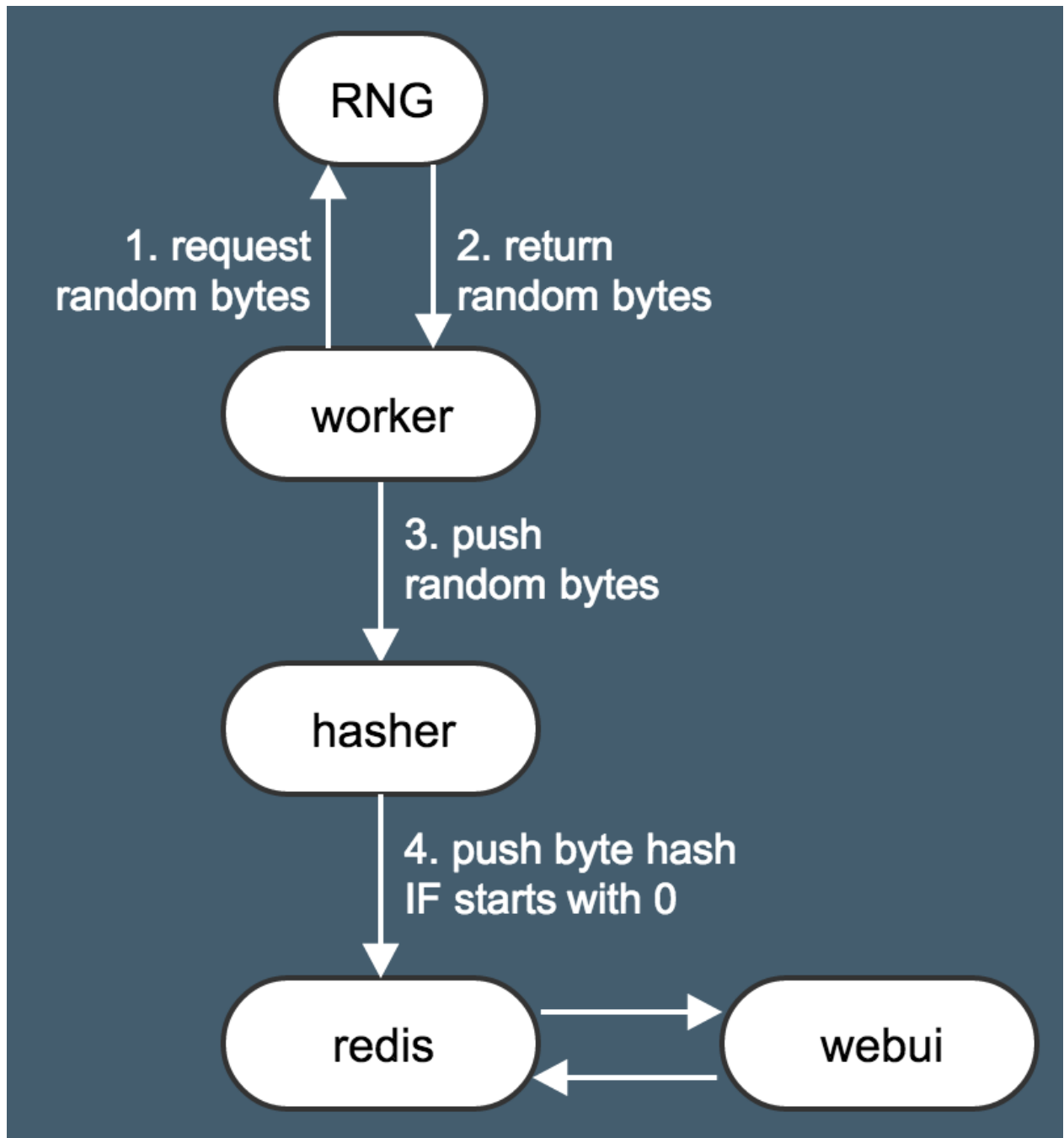
- Deploy an application orchestrated by either Swarm or Kubernetes via UCP
- Identify analogous orchestration components between the two orchestrators

Pre-requisites:

- UCP installed with 2 worker nodes (or more).
- No existing services running. Before beginning this exercise, delete all your previous services that have been deployed, in order to avoid port conflicts.

7.1 Background info

The application that we are going to deploy is a toy Dockercoin miner that hashes random numbers and logs any hash beginning with a 0 as a 'Dockercoin', and reports real-time results in a web UI. The app consists of five services, that interact like this:



7.2 Deploying to Swarm

The Swarm services that make up our app can be defined in a Compose file and then deployed in UCP via the web UI.

1. In the UCP web UI, navigate **Shared Resources** -> **Stacks** -> **Create Stack**.
2. Under 1. *Configure Application*, specify:
 - *Name*: dockercoins
 - *Orchestrator Mode*: **Swarm Services**
 - *Application File Mode*: **Compose File**

Click **Next**.

3. In the 2. *Add Application File* form, paste the following Compose file into the **docker-compose.yml** field:

```
version: "3.1"

services:
  rng:
    image: training/dockercoins-rng:1.0
    networks:
      - dockercoins
    ports:
      - "8001:80"
  hasher:
    image: training/dockercoins-hashier:1.0
    networks:
      - dockercoins
    ports:
      - "8002:80"
  webui:
    image: training/dockercoins-webui:1.0
    networks:
      - dockercoins
    ports:
      - "8000:80"
  redis:
    image: redis
    networks:
      - dockercoins
  worker:
    image: training/dockercoins-worker:1.0
    networks:
      - dockercoins

networks:
  dockercoins:
```

4. Click **Create**, wait for the services to deploy and then click **Done**. Dockercoins should now be listed on the **Shared Resources -> Stacks** page.
5. Navigate **Shared Resources -> Stacks -> Dockercoins -> Services**. A list of all the services composing Dockercoins is produced. Wait for all services to report 1/1 in their 'Status' column.
6. Open a browser tab and see Dockercoins in action by navigating to port 8000 at the public IP of any of UCP node.

7.3 Scaling the application

We want more Dockercoins! To increase mining speed, we will scale the `worker` service so that we will have more containers to handle the workload.

1. Scale the `worker` service to run 2 containers. Find the option to do this by clicking on the service in UCP, and then clicking the gear in the top right to edit the service.
2. Return to the web UI after the new worker container has spun up. Your mining speed should have doubled.
3. Try re-deploying the Dockercoins application exactly as above, with the following modification to the Compose file:

```
worker:
  image: training/dockercoins-worker:1.0
  networks:
```

```
- dockercoins
deploy:
  replicas: 2
```

The end result is the same, but the scale of the worker service is captured right in the application definition, rather than having to manage it through UCP after deployment. Also notice that re-deploying an existing stack *updates* the stack.

4. Finally, remove the Dockercoins application via the **Shared Resources** -> **Stacks** menu in UCP.

7.4 Deploying to Kubernetes

1. Set up your stack similarly to how you did above when first deploying to Swarm, but this time, select **Kubernetes Workloads** in the *1. Configure Application* form. Another dropdown will appear asking you to choose a Kubernetes namespace; choose **default**. Finally, use this slightly modified Compose file:

```
version: "3.1"

services:
  rng:
    image: training/dockercoins-rng:1.0
    networks:
      - dockercoins
    ports:
      - "32769:80"
  hasher:
    image: training/dockercoins-hashier:1.0
    networks:
      - dockercoins
    ports:
      - "32770:80"
  webui:
    image: training/dockercoins-webui:1.0
    networks:
      - dockercoins
    ports:
      - "32768:80"
  redis:
    image: redis
    networks:
      - dockercoins
  worker:
    image: training/dockercoins-worker:1.0
    networks:
      - dockercoins

networks:
  dockercoins:
```

You'll notice this is exactly the same as the Compose file we used for deploying Dockercoins above, except now we're exposing ports ≥ 32768 , above Kubernetes' reserved range for nodePort services. Once configured, click **Create**, and you'll be sent back to the UCP dashboard.

2. Navigate **Kubernetes** -> **Pods**. One pod has been created for each object under the services key in your compose.yml.
3. Navigate **Kubernetes** -> **Controllers**. A ReplicaSet has been created for each microservice to perform keep-alive on the pods we saw in the last step, and a Deployment has been created to manage things like rolling

updates and rollback.

4. Navigate **Kubernetes -> Load Balancers**. A Kube ClusterIP service has been created for every service, to make ReplicaSets reachable internally to the cluster by DNS resolution of the service name defined in the `compose.yml`, exactly analogously to how service name resolution behaved in Swarm.
5. Also in the Load Balancers list, notice that all the microservices that exposed a public port have a `*-published` service, like `webui-published` and `rng-published`. These are Kubernetes LoadBalancer services that make ReplicaSets reachable from the external network, similar to Swarm's L4 mesh net.
6. Click on the `webui-published` Kube service. Scroll to the bottom to find the *Ports* description; the NodePort assigned to this service should be 32768 as defined in the `compose.yml`. Visit your Kube-managed dockercoins at `<ucp-node-0 public IP>:32768`.
7. Clean up your Kubernetes deployment by deleting your Dockercoins stack.

7.5 Deploying Application Packages

In the two examples above, we deployed compose files as stacks to Swarm and Kubernetes. Another way to create a stack on either orchestrator is to deploy an *application package*, which is a superset of a compose file that includes some metadata and configuration.

1. In UCP, navigate **Shared Resources -> Stacks -> Create Stack**. Fill in the **Configure Application** box as follows:
 - Name: `demoapp`
 - Orchestrator Mode: `Swarm Services`
 - Application File Mode: `App Package`
 - Click **Next**.
2. In the **Single-file Application Package** field that appears, paste in the following application package file, and click **Create**:

```
# This section contains your application metadata.
version: 0.1.0
name: hello
description: "an app file demo"
maintainers:
- name: moby
  email: "whale@docker.com"
targets:
  swarm: true
---
# This section contains the Compose file that describes your application services.
version: "3.2"
services:
  hello:
    image: training/http-echo:${version}
    command: ["-text", "${text}"]
    ports:
      - "${port}:5678"
---
# This section contains the default values for your application settings.
port: 5678
text: hello development
version: 2.1
```

Note: To do the same thing for Windows users, please run this in Powershell: `Invoke-WebRequest https://bit.ly/2Px0Xpg -UseBasicParsing -o apppackage.yml`. Then, upload the file with the **Upload Application Package File** button instead of pasting the file.

Note especially the second and third stanzas, between the `---` delimiters: the middle stanza is a compose file, using variables expressed like `${version}`. The values to be substituted into these variables at runtime are expressed in the last stanza. In this way, we can capture environment-generic compose files and environment-specific variables, all in the same version-controllable file.

3. On any node in your cluster, hit your app at `curl -4 localhost:5678`; it should respond with the echo text you provided in the configuration block above, `hello development`.
4. *Kubernetes Challenge*: try reconfiguring the application package file above to deploy to Kubernetes. Very little has to change; hint: remember that port exposures in a compose file get translated into NodePort services in Kubernetes, which are confined to the port range 32768-35535.
5. Tear down your stack as you did above.

7.6 Conclusion

In this exercise, you deployed an entire application once as Swarm services, and again as Kubernetes workloads, from a single compose file. Capturing application configuration in compose files is an important best practice for application reproducibility and versioning; the compose file provides a single source of truth that can be distributed via your existing version control strategy, and easily handed off between developers and operations teams. Compose files are made further portable and flexible when described as an application package file, which conveniently factors out configuration so the same compose file can be used across multiple environments.

8 Combining Collections and Kubernetes Namespaces

By the end of this exercise, you should be able to:

- Place UCP worker nodes in custom resource collections in order to govern their workloads via UCP RBAC
- Associate a Kubernetes namespace with a resource collection in order to confine deployments in that namespace to run only on worker nodes in that resource collection.

8.1 Creating Collections

1. Login to UCP via your admin account.
2. Navigate to **Shared Resources** -> **Collections**.
3. On the collection **Swarm** click **View Children**; then do the same on the **Shared** collection.
4. Click **Create Collection** and enter **Development** under *Collection Name*. Then click **Create**.
5. Create another sibling collection called **Production**.

8.2 Associating Worker Nodes with Collections

We want to place worker nodes into the two collections we created above.

1. Navigate to **Shared Resources** -> **Nodes** -> **ucp-node-0**; click on the gear in the top right, then click **Collection**.
2. Click **View Children** under *Shared*, and then **Select Collection** beside *Development* and finally click **Save**, to place **ucp-node-0** in this collection.
3. Follow the same procedure to move **ucp-node-1** into the **/Shared/Production** collection.

8.3 Creating Kubernetes Namespaces

It is a good practice in Kubernetes to divide applications by namespaces.

1. Navigate to **Kubernetes** -> **Namespaces** and click **Create**.

2. In the **Object YAML** field enter this content to create a production namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

3. Click **Create** to have UCP create the namespace.
4. You should already have a namespace called `development` from an earlier exercise; if not, create it now.

8.4 Associating Nodes with Namespaces

1. Navigate to **Kubernetes -> Namespaces**.
2. Hover over the `development` namespace; click the ... that appears on the right of the table row, and then click **Link Nodes in Collection**
3. Navigate down to `Swarm/Shared/` and click **Select Collection** beside `Development`. The table at the bottom should now show only `ucp-node-0` if everything has been set up correctly; if so, click **Confirm**. Kube workload in the `development` namespace will now be confined to `ucp-node-0`.
4. Associate `ucp-node-1` with the `production` namespace by the same method.

8.5 Creating a Deployment in the development namespace

1. Navigate to **Kubernetes -> Namespaces**.
2. Hover over `development` and click **Set Context**.
3. Navigate **Kubernetes -> Create**.
4. In the *Object YAML* field enter this content to create a deployment:

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: development
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

5. Click **Create**.

This will create a Kubernetes Deployment object as well as a corresponding ReplicaSet managing 3 Pods each containing an instance of `nginx`. All these objects will live in the `development` namespace.

6. Navigate to **Kubernetes -> Controllers**. You should see the `nginx-deployment` we just created above.
7. Navigate to **Kubernetes -> Pods** and make sure they are all deployed on node `ucp-node-0` (this is the only worker node associated with collection `/Shared/Development` and linked to the `development` namespace.).

8.6 Creating a Deployment in the production namespace

Repeat the previous exercise but this time for the namespace `production`. Use the following yaml that defines our production deployment object:

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: tomcat-prod-deployment
  namespace: production
spec:
  selector:
    matchLabels:
      app: tomcat
  replicas: 5
  template:
    metadata:
      labels:
        app: tomcat
    spec:
      containers:
        - name: tomcat
          image: tomcat:9.0
          ports:
            - containerPort: 8080
```

8.7 Cleaning up

1. Set the context to namespace `development`.
2. Navigate to **Kubernetes** -> **Controllers**.
3. Check the box beside the Deployment you just made and delete it (**Actions** -> **Remove**).
4. Do the same with the production deployment.
5. Delete the two Kubernetes namespaces `development` and `production`.
6. Move the two worker nodes back into the `/Shared` collection.

8.8 Conclusion

In this exercise we have created multiple Kubernetes resources in their own custom namespaces from within UCP. We have associated worker nodes with different namespaces and as such made sure that the corresponding pods were only deployed to the designated worker nodes. Fluency with Kubernetes namespaces and RBAC for nodes in UCP is useful when managing multiple environments on the same cluster; we could imagine running development and testing versions of the same Kubernetes deployments on the same cluster, and prevent them from colliding in their label associations, DNS lookups and host resources by putting them in separate namespaces associated with separate nodes.

9 Basic Swarm Routing Models

By the end of this exercise, you should be able to:

- Route traffic to a swarm service using the correct tool in each of the following scenarios:
- Cluster-internal traffic to a stateless service
- Cluster-internal traffic to a stateful service
- Ingress traffic to a stateless service
- Ingress traffic to a stateful service

9.1 Routing Cluster-Internal Traffic

By *cluster-internal traffic*, we mean traffic from originating from a container running on your swarm, sending a request to another container running on the same swarm. We need to consider two cases:

- **Stateless services:** can be load balanced across freely. A containerized API would be a common example.
- **Stateful services:** we may need to make explicit and consistent decisions about what container we send a request to. A containerized database using simple local storage is a common example.

9.1.1 Routing to Stateless Services

Routing to stateless services internal to a swarm is the simplest form of swarm routing: service discovery will be provided by DNS lookup, and Docker will ensure round-robin load balancing across destination containers. All you must do as an operator is ensure the communicating services are connected to the same overlay network as follows.

1. On `ucp-manager-0`, create an overlay network for containers to communicate on:

```
[centos@ucp-manager-0 ~]$ docker network create --driver overlay demonet
```

2. Declare a service to route traffic to; call it `destination`:

```
[centos@ucp-manager-0 ~]$ docker service create --name destination \
  --network demonet \
  --replicas 3 \
  training/whoami:latest
```

3. Declare another service we'll use to send traffic to the first:

```
[centos@ucp-manager-0 ~]$ docker service create --name origin \
  --network demonet \
  nicolaka/netshoot sleep 1000000
```

4. Use `docker service ps origin` to find the node that the single replica of your origin service got scheduled on, and connect to that node. I'll refer to this as `node-x` below.

5. On `node-x`, use `docker container ls | grep origin` to find the container ID of your origin container, and create a bash shell within that container:

```
[centos@node-x ~]$ docker container exec -it <container ID> bash
```

6. curl your destination service by name 4 times:

```
bash-4.4# curl destination:8000
I'm 9b2c603dddea
bash-4.4# curl destination:8000
I'm 9465b262a410
bash-4.4# curl destination:8000
I'm 15c091a4a796
bash-4.4# curl destination:8000
I'm 9b2c603dddea
```

Docker's DNS resolves `destination` to the VIP of the destination service, and round-robin load balances the request across the corresponding containers.

7. Use `nslookup` to see exactly what `destination` is resolving to:

```
bash-4.4# nslookup destination

Server:      127.0.0.11
Address:     127.0.0.11#53

Non-authoritative answer:
```

```
Name: destination
Address: 10.0.1.56
```

That Address: 10.0.1.56 is the virtual IP of your destination service. Sending traffic there tells Docker to round-robin load balance across containers as you saw above.

8. Remove your destination service (we'll use origin again later, leave it alone):

```
[centos@ucp-manager-0 ~]$ docker service rm destination
```

9.1.2 Routing to Stateful Services

If our destination service is stateful, the automatic round-robin load balancing provided by VIP resolution above isn't appropriate; we want to be able to discover the IPs of the containers underlying a service directly, and make our own routing decisions from there.

1. Create a new destination service, this time in *DNSRR mode*:

```
[centos@ucp-manager-0 ~]$ docker service create --name destination \
  --network demonet \
  --replicas 3 \
  --endpoint-mode dnsrr \
  training/whoami:latest
```

2. Connect to your origin container exactly as you did above. Run `nslookup` to see what destination resolves to now:

```
bash-4.4# nslookup destination

Server:      127.0.0.11
Address:     127.0.0.11#53

Non-authoritative answer:
Name:   destination
Address: 10.0.1.50
Name:   destination
Address: 10.0.1.49
Name:   destination
Address: 10.0.1.51
```

The three IPs at the bottom (10.0.1. [49–51] in my case) are the IPs on demonet of your three destination containers. From here, you can choose which one you'd like to route to; try curling one of them a few times, and see that your request always goes to the same destination container.

3. Clean up your services:

```
[centos@ucp-manager-0 ~]$ docker service rm origin destination
```

9.2 Routing Ingress Traffic

In the case that we want to allow traffic in to a service from the outside world, we need to route traffic from a port in the host's network namespace onto a port in the container's network namespace; both methods below do essentially this, through different means.

9.2.1 Ingress Traffic to Stateless Services

In the case of stateless services, we want to leverage Docker's built-in VIP load balancing that we saw in the first section above. To do so, we'll forward traffic from a port on every host in our swarm to an IP Virtual Server provided

by the Docker Engine, which will automatically forward the traffic to the appropriate VIP and service. This is called swarm's *L4 routing mesh*.

1. Re-create your destination service with a cluster-wide port exposure:

```
[centos@ucp-manager-0 ~]$ docker service create --name destination \
  --network demonet \
  --replicas 3 \
  -p 8080:8000 \
  training/whoami:latest
```

This will make Docker listen on port 8080 of every host in the swarm, and load balance traffic received there to destination's containers on their internal port 8000.

2. From your infra node (or just your local machine if possible), contact your exposed destination service:

```
[centos@infra ~]$ curl <ucp-manager-0 public IP>:8080
I'm 01ca0ea0f69f
[centos@infra ~]$ curl <ucp-manager-0 public IP>:8080
I'm 358b7633339f
[centos@infra ~]$ curl <ucp-manager-0 public IP>:8080
I'm 73c88db127f2
```

The same round robin load balancing as above, but this time for traffic originating outside your cluster. Try replacing `<ucp-manager-0 public IP>` with the public IP of a node in your cluster that does *not* have a destination container running on it; everything will still work, since it's Docker itself listening on port 8080 on every host in the swarm, not the containers directly. The IPVS will redirect traffic across nodes using the ingress overlay network, regardless of which node the traffic originally arrived on.

3. Remove your destination service as usual.

9.2.2 Ingress Traffic to Stateful Services

Just as with routing traffic originating internally to the swarm, if our target service is stateful, we don't want Docker to interfere and load balance traffic across containers; we want to make that decision ourselves. We can do so by mapping host ports directly onto container ports, circumventing the IPVS.

1. Recreate your destination service one last time, this time using `--publish mode=host`:

```
[centos@ucp-manager-0 ~]$ docker service create --name destination \
  --network demonet \
  --replicas 3 \
  --publish mode=host,target=8000 \
  training/whoami:latest
```

2. Docker will map a random available host port onto port 8000 inside each container started for the destination service. Find one such host port (32768 in all cases in this example):

```
[centos@ucp-manager-0 ~]$ docker service ps destination
... NAME                IMAGE                NODE                ... PORTS
... destination.1       training/whoami:latest ucp-manager-0       ... *:32768->8000/tcp
... destination.2       training/whoami:latest ucp-manager-1       ... *:32768->8000/tcp
... destination.3       training/whoami:latest ucp-manager-2       ... *:32768->8000/tcp
```

3. From infra, curl the public IP and port you found above (determine the public IP from the NODE column above), for example:

```
[centos@infra ~]$ curl 54.198.130.41:32768
I'm 929a4b4edd03
[centos@infra ~]$ curl 54.198.130.41:32768
I'm 929a4b4edd03
```

```
[centos@infra ~]$ curl 54.198.130.41:32768
I'm 929a4b4edd03
```

The same container ID is returned every time, since (in my case) port 32768 on the swarm node at the public IP I gave is getting forwarded directly to the destination container hosted on that node.

4. Delete your destination service one last time.

9.3 Conclusion

In this exercise, you set up basic examples of swarm routing intended for use with stateful and stateless services receiving traffic from inside or outside your swarm. Understanding how traffic is meant to be routed to your services is a key thing to find out from your developers as images move from development into production; ask them for clear answers on whether their containers are stateful or stateless, and whether they expect traffic from inside the swarm or from the external network, and then follow the examples above to successfully configure traffic routing as designed.

10 Basic Kubernetes Routing Models

By the end of this exercise, you should be able to:

- Route traffic to a kube deployment using the correct tool in each of the following scenarios:
- Cluster-internal traffic to a stateless deployment
- Cluster-internal traffic to a stateful deployment
- Ingress traffic to a stateless deployment

10.1 Routing Cluster-Internal Traffic

By *cluster-internal traffic*, we mean traffic from originating from a pod running on your cluster, sending a request to another pod running on the same cluster. We need to consider two cases:

- **Stateless deployments:** can be load balanced across freely. A containerized API would be a common example.
- **Stateful deployments:** we may need to make explicit and consistent decisions about what container we send a request to. A containerized database using simple local storage is a common example.

10.1.1 Routing to Stateless Deployments

Routing to stateless deployments internal to a cluster relies on use of a `ClusterIP` service, which will provide a stable networking endpoint for a collection of containers, similar in usage (though not implementation) to Swarm's VIPs.

1. Declare a deployment to route traffic to by navigating on UCP **Kubernetes** -> **Create**, pasting in the following yaml and clicking **Create**:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: destination
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: dest
  template:
    metadata:
      labels:
```

```

    app: dest
  spec:
    containers:
    - name: who
      image: training/whoami:latest

```

2. Declare a ClusterIP service to route traffic to your destination deployment internally:

```

apiVersion: v1
kind: Service
metadata:
  name: destination-entripoint
  namespace: default
spec:
  selector:
    app: dest
  ports:
  - port: 8080
    targetPort: 8000

```

This service will route traffic sent to port 8080 on the IP generated by this service, to port 8000 in pods which match the `app: dest` label selector.

3. Declare another deployment we'll use to send traffic to the first:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: origin
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: orig
  template:
    metadata:
      labels:
        app: orig
    spec:
      containers:
      - name: probe
        image: nicolaka/netshoot:latest
        command: ["sleep"]
        args: ["1000000"]

```

4. Next we'd like to open a terminal inside the `netshoot` container of our `origin` pod:

- Navigate **Kubernetes** -> **Pods** -> **origin**
- Click on the `**>_` icon in the top right
- Enter `bash` in the box and click **Run**

5. At the terminal you just opened in UCP, use `nslookup` to see what `destination-entripoint` resolves as:

```

bash-4.4# nslookup destination-entripoint

Server:          10.96.0.10
Address:         10.96.0.10#53

Name:   destination-entripoint.default.svc.cluster.local

```



```
Address: 10.96.231.233
```

The Address on the last line is the cluster IP chosen for this service. Traffic to port 8080 (which we defined in the yaml for our service above) at this IP will get randomly load balanced across pods matching the label selector provided.

6. In the same terminal, try curling the service name and port:

```
bash-4.4# curl destination-entriypoint:8080
I'm destination-6747cb6454-99xzs
bash-4.4# curl destination-entriypoint:8080
I'm destination-6747cb6454-7h6jg
bash-4.4# curl destination-entriypoint:8080
I'm destination-6747cb6454-7h6jg
bash-4.4# curl destination-entriypoint:8080
I'm destination-6747cb6454-7h6jg
bash-4.4# curl destination-entriypoint:8080
I'm destination-6747cb6454-sdpgr
```

Unlike swarm, a kube ClusterIP routes traffic randomly to pods it serves, rather than round robin.

7. Delete your 'destination-entriypoint' service:
 - Navigate **Kubernetes -> Load Balancers**
 - Check the checkbox beside **destination-entriypoint**
 - Click **Actions -> Remove**

10.1.2 Routing to Stateful Deployments

If our destination deployment is stateful, the random load balancing provided by the ClusterIP service above isn't appropriate; we want to be able to discover the IPs of the pods underlying a deployment directly, and make our own routing decisions from there.

1. Create a new destination-entriypoint service, this time as a *headless ClusterIP*:

```
apiVersion: v1
kind: Service
metadata:
  name: destination-entriypoint
  namespace: default
spec:
  clusterIP: None
  ports:
  - port: 8080
  selector:
    app: dest
```

Note the clusterIP: None line - that's what makes this service headless. Also, understand that the port: 8080 key here has no effect for headless services, but is required for API validation; this has been patched upstream and won't be required in later versions of Kubernetes and UCP.

2. Open a bash terminal inside the netshoot container of our origin pod like you did above, and try nslookup on destination-entriypoint again:

```
bash-4.4# nslookup destination-entriypoint

Server:           10.96.0.10
Address:          10.96.0.10#53

Name:   destination-entriypoint.default.svc.cluster.local
Address: 192.168.162.197
```

```
Name: destination-entripoint.default.svc.cluster.local
Address: 192.168.197.132
Name: destination-entripoint.default.svc.cluster.local
Address: 192.168.107.70
```

This time, the IPs of all the pods matched by your service's label selector are returned. Try curling a few of these on port 8000 to confirm the routing works.

3. Delete your destination-entripoint service as above.

10.2 Routing Ingress Traffic

In the case that we want to allow traffic in to a deployment from the outside world, we need to route traffic from a port in the host's network namespace onto a port in the pod's network namespace; we can accomplish this through a NodePort service.

10.2.1 Ingress Traffic to Stateless deployments

In the case of stateless deployments, we want to build off of the ClusterIP service that we saw in the first section above. To do so, we'll create a NodePort service which will forward traffic from a random port on every host in our cluster to a ClusterIP service it automatically creates for us.

1. Re-create your destination-entripoint service as a NodePort service:

```
apiVersion: v1
kind: Service
metadata:
  name: destination-entripoint
  namespace: default
spec:
  type: NodePort
  selector:
    app: dest
  ports:
    - port: 8080
      targetPort: 8000
```

Key values here being the type: NodePort to declare this as the desired service type, and the port and targetPort keys which have the same meaning as the ClusterIP service we created in the first exercise above.

2. Navigate **Kubernetes -> Load Balancers -> destination-entripoint**, scroll to the bottom, and find the **Node Port** entry in the table (should be something in the 32768-35535 range). This is the port your deployment pods are reachable at, on any IP in your kube cluster. Try curling : from your infra node to confirm the traffic is routed as expected.
3. Delete your origin, destination, and destination-entripoint deployments and services as above.

10.3 Conclusion

In this exercise, you set up basic examples of cluster routing intended for use with stateful and stateless deployments receiving traffic from inside or outside your cluster. Understanding how traffic is meant to be routed to your deployments is a key thing to find out from your developers as images move from development into production; ask them for clear answers on whether their containers are stateful or stateless, and whether they expect traffic from inside the cluster or from the external network, and then follow the examples above to successfully configure traffic routing as designed. One configuration that is not currently easily supported on Kubernetes is routing traffic to specific (ie stateful) pods from outside the cluster in a way analogous to Swarm's mode=host binding; nevertheless, we'll see in a later exercise some more sophisticated kube routing technology that will solve this problem in some cases.

11 L7 Swarm Routing with Interlock

By the end of this exercise, you should be able to:

- Set up Interlock as part of a UCP deployment
- Configure Swarm services to accept ingress traffic via hostname header
- Configure cookie-based sticky sessions for services routed at L7 by Interlock

11.1 Enabling Interlock

1. From a UCP admin account, navigate **admin** -> **Admin Settings** -> **Layer 7 Routing**.
2. Leave the default ports alone (80 for **HTTP** and 8443 for **HTTPS**), and tick the checkbox which says **Enable Layer 7 Routing**.
3. Click **Save**, at the bottom.
4. Click the **X** in the top right to escape from this config page, then navigate **Swarm** -> **Networks**. There should be a network named `ucp-interlock`.
5. Navigate **Swarm** -> **Services** to display all your swarm services. There should be one for Interlock, one for extension and one for proxy:

Note: If you don't see these services, click the gear icon (in the service view page) and check **Show system resources**.

3 Service(s)						
	STATUS	NAME	IMAGE	MODE	UPDATED AT	LAST ERROR
<input checked="" type="checkbox"/>	1/1	ucp-interlock	dockereng/ucp-interlock:3.1.0-tp8	Replicated	2 minutes ago	
<input checked="" type="checkbox"/>	1/1	ucp-interlock-extension	dockereng/ucp-interlock-extension:3.1.0-tp8	Replicated	2 minutes ago	
<input checked="" type="checkbox"/>	2/2	ucp-interlock-proxy	dockereng/ucp-interlock-proxy:3.1.0-tp8	Replicated	2 minutes ago	

Note the proxy service has two replicas, for high availability.

11.2 Deploying Services with Interlock

1. On `ucp-manager-0`, create an overlay network for our service:

```
[centos@ucp-manager-0 ~]$ docker network create -d overlay demo
```

2. Create and publish a service, making sure to use the two labels `com.docker.lb.hosts` and `com.docker.lb.port`:

```
[centos@ucp-manager-0 ~]$ docker service create --name demo \
  --network demo \
  --label com.docker.lb.hosts=demo.A \
  --label com.docker.lb.port=8000 \
  training/whoami:latest
```

The `com.docker.lb.hosts` label tells Interlock the value of the Host header that should be routed to this service, and `com.docker.lb.port` tells the proxy service to which container port to send the request to.

3. Make sure the service is up and running by curling it from your `infra` node, outside your cluster:

```
[centos@infra ~]$ curl -H "Host: demo.A" http://<ucp-manager-0 public IP>
I'm 2c206a3a4741
```

The request has been routed to the container based on the Host value in the request header.

4. Scale the service to four replicas:

```
[centos@ucp-manager-0 ~]$ docker service update --replicas 4 demo
```

5. See that traffic is load balanced across all our new service replicas:

```
[centos@infra ~]$ for N in `seq 1 10`; \
do curl -H "Host: demo.A" http://<ucp-manager-0 public IP>; \
done;
```

Which produces something like:

```
I'm 2c206a3a4741
I'm 2c206a3a4741
I'm cf659f3e6093
I'm cf659f3e6093
I'm 89598836e0d1
I'm 89598836e0d1
I'm d83e3b1acea6
I'm d83e3b1acea6
I'm 2c206a3a4741
I'm 2c206a3a4741
```

As you can see, the instance IDs are being displayed twice before proceeding to the next one. This is because the L7 load balancer round robins across both the replicas of the proxy service, and each of those round robins across the four replicas of the demo service.

6. List Docker's config files:

```
[centos@ucp-manager-0 ~]$ docker config ls
```

ID	NAME	...
rsmr2wluw42j9y7bqtpjbr7qc	com.docker.interlock.extension.f24ce8	...
tzcdlfdlro4bpyqj5dve98gn3	com.docker.interlock.proxy.955595	...
qf2zpycobvi3ve07ojpbzs2ap	com.docker.interlock.proxy.ce9f9a	...
zd3n46stobvzjvwqeyrr44jj3	com.docker.license-0	...
rqpz7kj20zyxhstq086t0gecq	com.docker.ucp.interlock.conf-1	...
ku4cmjk7cfm3i42a87bzzndmx	com.docker.ucp.internal-config.8FSC988ACA0PUAAUV5QANUM7	...

The ones that look like `com.docker.interlock.proxy.*` are the nginx configuration files created for the proxy by the extension service.

7. Inspect the *most recent* config file (note there are two: the newest and the second-newest; we want the newest):

```
[centos@ucp-manager-0 ~]$ docker config inspect --pretty <newest proxy config ID>
```

There should be a block that looks similar to:

```
upstream up-demo.local {
    zone up-demo.local_backend 64k;

    server 10.0.3.39:8000;
    server 10.0.3.37:8000;
    server 10.0.3.38:8000;
    server 10.0.3.31:8000;
}
```

Those are the task IPs for each of your four service tasks, to which Interlock's proxy is forwarding traffic.

8. Clean up by removing your demo service.

11.3 Configuring Sticky Sessions

In case we want a client to always be routed to the same backend container by Interlock, we can configure our service to require *sticky sessions* as follows.

1. Create an L7-routed service with sticky sessions:

```
[centos@ucp-manager-0 ~]$ docker service create --name stickydemo \
  --network demo \
  --replicas 4 \
  --label com.docker.lb.hosts=demo.sticky \
  --label com.docker.lb.port=8000 \
  --label com.docker.lb.sticky_session_cookie=session \
  training/whoami:latest
```

2. Hit your new service a bunch of times from infra, passing in an arbitrary value for the session cookie:

```
[centos@infra ~]$ for N in `seq 1 10`; \
do curl -b session=mycookie \
  -H "Host: demo.sticky" http://<ucp-manager-0 public IP>; \
done;

I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
I'm 1402abfc65c9
```

The request gets sent to the same backend every time. Change the value of `mycookie` to a different string and try again a few times, until you see your requests getting routed to a few different backend containers.

3. Clean up by removing your `stickydemo` service and `demo` network.

11.4 Conclusion

In this exercise, we set up Interlock 2 in UCP, and used it to route external requests to a service based on the Host header found therein, either in round robin fashion for stateless services, or with cookie-based sticky sessions for stateful services. While similar results can technically be achieved with L4 routing, this would require a separate configuration in an external load balancer for each service expecting ingress traffic at L4. L7 and Interlock make it possible to configure just one point of ingress in your external load balancer that passes traffic onto Interlock, which then handles routing the traffic to its destination.

12 Kubernetes Ingresses

Kubernetes Ingress objects allow more sophisticated routing patterns to be established for traffic originating outside your cluster. By the end of this exercise, you should be able to:

- Set up a nginx-based Kubernetes IngressController

- Configure L7 routing, path based routing and sticky sessions with a Kubernetes Ingress object

12.1 Setting up an IngressController in UCP

Before we can create any Ingress objects, we need an IngressController to manage them and provide the actual proxy to do the routing; we'll set up an nginx-based IngressController.

1. First we need a Kubernetes *service account* for our IngressController, to allow it to automatically make calls to the Kube API to find out the scheduling and networking configuration it will need. Start by creating a namespace for the IngressController and its service account (remember, you can create any Kube object from its yaml by pasting it into the box under **Kubernetes** -> **Create** in UCP):

```
apiVersion: v1
kind: Namespace
metadata:
  name: ingressnginx
```

2. Create a new grant that gives the default service account restricted access to all Kubernetes namespaces:
 1. Navigate **Access Control** -> **Grants** -> **Create Role Binding**
 2. Under **Subject** select **SERVICE ACCOUNT**, and then select Namespace **ingressnginx** and Service Account **default**. Click **Next**.
 3. Under **Resource Set**, enable the toggle labeled **Apply Role Binding to all namespace (Cluster Role Binding)**, and click **Next**.
 4. Under **Role** select **ucp.restricted-control**.
 5. Click **Create**.
3. Deploy your nginx IngressController by navigating **Kubernetes** -> **Create** and pasting in the yaml found at <https://bit.ly/2QZA9qL>, and clicking **Create**.
4. Confirm that everything deployed correctly by first setting your UCP context to the **ingressnginx** namespace, then checking your deployments:
 - Navigate **Kubernetes** -> **Namespaces**, hover over *ingressnginx*, and click **Set Context**.
 - Navigate **Kubernetes** -> **Controllers**. If all is well, you should see 1/1 pods running for **default-http-backend** and **nginx-ingress-controller**, both Deployments and ReplicaSets.
5. Under **Kubernetes** -> **Load Balancers**, there should be a NodePort service called **ingressnginx**; this is the external point of ingress to your IngressController. Find its public HTTP port from your **infra** node (make sure you've got a client bundle configured there so you can reach your cluster):

```
[centos@infra ~]$ kubectl describe -n ingressnginx service ingressnginx
```

```
Name:                ingressnginx
Namespace:           ingressnginx
Labels:              <none>
Annotations:         <none>
Selector:            app=ingressnginx
Type:                NodePort
IP:                  10.96.40.234
Port:                http 80/TCP
TargetPort:          80/TCP
NodePort:            http 33151/TCP
Endpoints:           192.168.123.201:80
Port:                https 443/TCP
TargetPort:          443/TCP
NodePort:            https 35344/TCP
Endpoints:           192.168.123.201:443
Session Affinity:    None
```

```
External Traffic Policy: Cluster
Events:                  <none>
```

In this example, the public HTTP port is 33151 - you'll need this later, when you want to connect to a Kubernetes service via your IngressController.

12.2 Configuring L7 and Path-Based Routing

1. Start by creating two deployments, each with a ClusterIP service pointing at them which we'll route to at L7 and by path. The ClusterIP services are named `who-1` (on port 3000), and `who-2` (on port 3100). yml to create these resources is available at <https://bit.ly/2NRf6ES>.
2. We are now ready to define **Ingress** resources that defines routing rules for the two services:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: who-ingress
  namespace: default
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: who-1
          servicePort: 3000
      - path: /bar
        backend:
          serviceName: who-2
          servicePort: 3100
```

This will define and configure the IngressController with the route mapping :

- `example.com/foo` —> service “who-1”
- `example.com/bar` —> service “who-2”

3. In your terminal use `curl` to test the routing, where `<public IP>` is the public IP address of any cluster node and `<port>` is the NodePort of the IngressController service as discussed further up (in my example it was 33151):

```
[centos@infra ~]$ curl -H "Host: example.com" <public IP>:<port>/foo
I'm who-1-67f887d79f-qq2tk

[centos@infra ~]$ curl -H "Host: example.com" <public IP>:<port>/bar
I'm who-2-69ddd74897-qbddp
```

Repeat the `curl`s a few times to see that `/foo` always gets routed to the same place, as does `/bar`.

12.3 Configuring Sticky Sessions

1. On UCP, navigate **Kubernetes** -> **Load Balancers**, and delete your `who-ingress` Ingress from the last step (remember, it's in the default Kubernetes namespace; if you can't find it in the load balancers list, look under the **Namespaces** tab and click **Set Context** while hovering over the namespace you want).

2. Navigate **Kubernetes** -> **Controllers** -> **who-1**, and click the gear in the top right to edit your deployment. Set replicas: 3 so we can prove to ourselves that our sticky sessions are responsible for sending us to the same pod every time.
3. Create a new ingress via the following yaml:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: sticky-who
  namespace: default
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "route"
    nginx.ingress.kubernetes.io/session-cookie-hash: "sha1"
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: who-1
          servicePort: 3000
```

This instructs Kube to provide and check the value of the route cookie in a request, and route the request to the who-1 pod corresponding to the value of that cookie.

4. The Ingress we've created will automatically generate sticky session tokens on request; use curl to record a sticky session cookie (this would be done automatically by a browser, but we need to manage it by hand with curl), where <public IP> is the public IP address of any cluster node and <port> is the NodePort of the IngressController:

```
[centos@infra ~]$ curl -c stickycookie -H "Host: example.com" <public IP>:<port>/foo
```

5. Hit your who-1 deployment repeatedly from infra, passing in the cookie you just got:

```
[centos@infra ~]$ for N in `seq 1 10`; \
do curl -b stickycookie -H "Host: example.com" <public IP>:<port>/foo; \
done;

I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
I'm who-1-67f887d79f-mltg2
```

The cookie ensures the request gets sent to the same pod, every time.

6. Clean up by deleting your sticky-who service and your who-1 and who-2 deployments.

12.4 Conclusion

In this exercise, we set up a Kubernetes IngressController and used it to manage several different Ingress objects, which provided advanced routing capabilities like L7 routing and cookie-based sticky sessions. As in the Swarm and Interlock case, an IngressController essentially provides a managed reverse proxy which is automatically reconfigured as needed. While Interlock relies on a Swarm service to monitor scheduling decisions, a Kube IngressController uses a service account to access the Kube API and learn what it needs to reconfigure its proxy accordingly.

13 Release Models in Swarm

By the end of this exercise, you should be able to:

- Configure an external load balancer to route traffic to the swarm mesh net for canary and blue / green releases.

13.1 Blue / Green Releases

In a blue / green release, we maintain two versions of our app simultaneously: the current production version (call that the 'green' version) which our external load balancers route traffic to, and the next version we're considering rolling out to production (call that the 'blue' version) which is currently not routed to in production. Once we're satisfied that the blue version is ready for release, we switch our load balancer to point at the blue version instead of the green, and their roles reverse: blue is now in production, and green can be updated and tested safely.

1. Deploy version 1.0 of your app as a swarm service on the L4 mesh at port 8000, and version 2.0 on port 8001:

```
[centos@ucp-manager-0 ~]$ docker service create \
  -p 8000:80 \
  --name current_production \
  training/petpic:1.0

[centos@ucp-manager-0 ~]$ docker service create \
  -p 8001:80 \
  --name production_candidate \
  training/petpic:2.0
```

Visit any public IP in your cluster on port 8000 and 8001, and confirm you can see versions 1.0 (goats) and 2.0 (cats) of our app, respectively.

2. On your infra node, create a load balancer config file called `worker.lb.conf`:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

stream {
    upstream ucp_workers {
        server <ucp-node-0 public IP>:8000;
        server <ucp-node-1 public IP>:8000;
    }
    server {
        listen 80;
        proxy_pass ucp_workers;
    }
}
```

```
}
}
```

This will load balance traffic arriving at your proxy's port 80 across port 8000 on each of your UCP worker, where version 1.0 of our app is currently being served - the 'green' version of our deployment.

3. Make sure your infra node's Docker client is pointing at its own Docker engine (and not at the UCP cluster via the client bundle you were using earlier):

```
[centos@infra ~]$ unset DOCKER_TLS_VERIFY COMPOSE_TLS_VERSION \
  DOCKER_CERT_PATH DOCKER_HOST
```

4. Start a containerized load balancer on your infra node, sending traffic to your 'green' deployment:

```
[centos@infra ~]$ docker container run -d --name worker-lb \
  --restart=unless-stopped \
  -p 8000:80 \
  -v ${PWD}/worker.lb.conf:/etc/nginx/nginx.conf:ro \
  nginx:stable-alpine
```

Visit your 1.0 deployment via your load balancer at <infra public IP>:8000 to confirm all is working well.

5. Once you're ready to switch from your 'green' to your 'blue' deployment, open `worker.lb.conf` and change port 8000 in the `ucp_workers` block (where your 'green' environment is accepting ingress) to port 8001 (where your 'blue' environment is listening).
6. Next you'll need to reload nginx. Start a shell in your load balancer container:

```
[centos@infra ~]$ docker container exec -it worker-lb sh
```

7. Reload nginx, and list processes:

```
/ # nginx -s reload
2018/10/19 16:27:15 [notice] 69#69: signal process started

/ # ps
PID   USER     TIME   COMMAND
   1   root      0:00   nginx: master process nginx -g daemon off;
  59   root      0:00   sh
  66   nginx     0:00   nginx: worker process is shutting down
  70   nginx     0:00   nginx: worker process
  71   root      0:00   ps
```

You may or may not see an nginx process with the message `worker process is shutting down` like above. If you don't, that's ok, no action is needed. If you do, kill it with `kill <PID>`, where <PID> can be found in the leftmost column of the `ps` output (66 in the example above).

8. Refresh your browser's connection to <infra public IP>:8000. You should see a picture of a cat, served by version 2.0 of our app - the 'blue' version of our production environment, to which your load balancer is now sending all traffic. (If you still see the goats, try a hard refresh of your browser - the goats picture might be sitting in your browser's cache).

13.2 Canary Releases

While blue / green deployments rely on internal testing of the offline environment, a *canary deployment* is designed to gather user feedback on updates while minimizing risk. To do this, we configure an external load balancer to route only a small fraction of traffic to our canary deployment, while the bulk of traffic is sent to our stable deployment.

1. In your load balancer configuration `worker.lb.conf`, change the upstream block to look like this:

```
upstream ucp_workers {
    server <ucp-node-0 public IP>:8000 weight=4;
```

```
server <ucp-node-1 public IP>:8000 weight=4;
server <ucp-node-0 public IP>:8001 weight=1;
server <ucp-node-1 public IP>:8001 weight=1;
}
```

This instructs nginx to send 8 out of every 10 requests to our 1.0 deployment on port 8000 (split evenly 4 and 4 across our two workers), and the other 2 out of 10 requests to our canary listening on port 8001, again split equally across our two workers.

2. Reload nginx as you did in the previous section.
3. Attempt to curl you load balancer's public port: `curl <infra public IP>:8000`. Do this 10 times; you should see 8 of the requests getting routed to the 1.0 version, and two to the 2.0 version.
4. Clean up everything by removing your `worker-lb` container on `infra`, and your `current_production` and `production_candidate` services from `ucp-manager-0`.

13.3 Conclusion

In this exercise, you saw simple examples of using an external load balancer to set up blue / green and canary releases of swarm services on the L4 mesh net. Note that this only applies to services exposed on the external mesh; there's no straightforward way to do something analogous for services communicating by VIP internally to your swarm, since a VIP only ever points to one set of identically configured containers, and can't be changed. If you want to achieve similar release models internally to your swarm, you need to introduce another degree of freedom in service discovery therein; for example, another service through which your application logic performs service discovery and which you can configure with more sophisticated routing rules, rather than relying on the swarm built-in DNS lookup.

14 Release Models in Kubernetes

Unlike Swarm, Kubernetes' *label selectors* and separation between service and deployment objects give us a built in mechanism to switch and load balance routing to our pods that does not rely on an external load balancer. By the end of this exercise, you should be able to:

- Configure Kubernetes label selectors for canary and blue / green releases of deployments.

14.1 Blue / Green Releases

1. Create 'blue' and 'green' deployments based on `training/petpic:1.0` and `:2.0` using the following yaml:

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: green
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: green
  template:
    metadata:
      labels:
        app: green
    spec:
      containers:
      - name: green
```

```

        image: training/petpic:1.0
        ports:
          - containerPort: 80
      ---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: blue
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: blue
  template:
    metadata:
      labels:
        app: blue
    spec:
      containers:
        - name: blue
          image: training/petpic:2.0
          ports:
            - containerPort: 80

```

2. Create a NodePort service that points to the 'green' deployment:

```

apiVersion: v1
kind: Service
metadata:
  name: bluegreen
  namespace: default
spec:
  type: NodePort
  selector:
    app: green
  ports:
    - port: 8080
      targetPort: 80

```

Note the selector: app: green section; this service is pointing exclusively at pods with the app: green label.

3. Find the public port selected by your NodePort service, and visit your 'green' deployment at <ucp-manager-0 public IP>:<NodePort>. You should see the goats of version 1.0 of your app.
4. Now let's flip the routing to our 'blue' deployment. In UCP, navigate **Kubernetes -> Load Balancers -> bluegreen**, and click the gear in the top-right to edit this NodePort's definition.
5. In the yaml presented, change selector: app: green to app: blue, and click **Save**.
6. Refresh your browser at <ucp-manager-0 public IP>:<NodePort>. The NodePort service now label-matches the 'blue' pods, and directs traffic there. (If the new version doesn't appear, try clearing your browser cache or using a private or 'incognito' historyless browser, or curling from the command line instead).
7. Remove your 'blue' and 'green' deployments and your 'bluegreen' service to clean up.

14.2 Canary Releases

1. Create 'production' and 'canary' deployments similar to the 'blue' and 'green' deployments above:

```

apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: production
  namespace: default
spec:
  replicas: 4
  selector:
    matchLabels:
      app: petpic
      stream: stable
  template:
    metadata:
      labels:
        app: petpic
        stream: stable
    spec:
      containers:
        - name: production
          image: training/petpic:1.0
          ports:
            - containerPort: 80
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: canary
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: petpic
      stream: canary
  template:
    metadata:
      labels:
        app: petpic
        stream: canary
    spec:
      containers:
        - name: petpic
          image: training/petpic:2.0
          ports:
            - containerPort: 80

```

Note how we've given our 'canary' deployment the same label `app: petpic` as our main 'production' deployment; this way, when we point a service at this label, the service will match and route to both 'production' and 'canary' pods. We've also added in an optional `stream` label - this is not actually necessary to make this example work, but it is a best practice which will let us tell 'production' and 'canary' pods apart by label if we need to for more sophisticated environments.

2. Create a NodePort service that matches the `app: petpic` label:

```

apiVersion: v1
kind: Service
metadata:
  name: canaryservice
  namespace: default
spec:
  type: NodePort
  selector:
    app: petpic
  ports:
    - port: 8080
      targetPort: 80

```

3. Find the NodePort port for your 'canaryservice' service, and curl it at <ucp-manager-0 public IP>:<NodePort>; you should see the HTML for either your 'production' or 'canary' deployments. Keep curling a bunch of times; the NodePort service will randomly spread your traffic across all the pods its label selector matches, which in this case should be the four 'production' pods and the one 'canary' pod (remember, it's *random* load balancing in this case, unlike the nginx defaults; you may have to curl more than 5 times to hit your canary).
4. Clean up by removing your 'production' and 'canary' deployments, and your 'canaryservice' service.

14.3 Conclusion

In this exercise, you set up blue / green and canary releases for Kubernetes deployments using label selection and kube services. In these examples we used NodePort services for convenience's sake, but the exact same pattern can be applied to ClusterIP services for doing blue / green and canary releases *internal* to your cluster in a way that was difficult to achieve on Swarm. The separation of services from pods and the flexibility of label selectors gives Kubernetes routing an extra degree of freedom that the rigid relationship between Swarm VIPs and tasks can't easily emulate.

15 Configuring Engine Logs

By the end of this exercise, you should be able to:

- Configure Docker Engine's logging driver
- Interpret the output of logs generated by the json-file and journald log drivers
- Configure log compression and rotation

15.1 Setting the Logging Driver

Docker offers a number of different logging drivers for recording the STDOUT and STDERR of PID 1 processes in a container; below we'll explore the defaults which correspond to the json-file driver, and the journald driver.

1. Run a simple container with the default logging configuration, and inspect its logs:

```

[centos@ucp-node-0 ~]$ docker container run -d centos:7 ping 8.8.8.8
[centos@ucp-node-0 ~]$ docker container logs <container ID>

PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=0.631 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=0.652 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=0.646 ms

```

2. Examine these same logs directly on disk; note <container ID> here is the full, untruncated container ID returned when you created the container above, or findable via `docker container ls --no-trunc`:

```
[centos@ucp-node-0 ~]$ sudo head -5 \
/var/lib/docker/containers/<container ID>/<container-ID>-json.log

{"log":"PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n","stream":"stdout","time":"2018-09-17T17:29:
{"log":"64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=0.631 ms\n","stream":"stdout","time":"2018-0
{"log":"64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=0.652 ms\n","stream":"stdout","time":"2018-0
{"log":"64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=0.646 ms\n","stream":"stdout","time":"2018-0
{"log":"64 bytes from 8.8.8.8: icmp_seq=4 ttl=113 time=0.577 ms\n","stream":"stdout","time":"2018-0
```

By default, logs are recorded as per the json-file driver format.

3. Configure your logging driver to send logs to the system journal by updating `/etc/docker/daemon.json` on `ucp-node-0` to look like this (note you'll need to open this file with `sudo` permissions in order to edit it):

```
{
  "storage-driver": "overlay2",
  "log-driver": "journald"
}
```

4. Restart Docker so the new logging configuration takes effect:

```
[centos@ucp-node-0 ~]$ sudo service docker restart
```

5. Run another container, just like the one you ran above, but this time name it `demo`:

```
[centos@ucp-node-0 ~]$ docker container run -d --name demo centos:7 ping 8.8.8.8
```

6. Inspect the system journal for messages from the `demo` container:

```
[centos@ucp-node-0 ~]$ journalctl CONTAINER_NAME=demo
```

In this way, container logs can be sent to the system journal for ingestion by a centralized logging framework along with the rest of the journal messages.

15.2 Configuring Log Compression and Rotation

By default, container logfiles can grow unbounded until all host disk is consumed. Many file-based logging drivers like `json-file` support automatic log rotation and compression.

1. Configure the Docker engine on `ucp-node-0` to create a json file of logs, swapping to a new file every 5 kb, preserving a maximum of 3 files, by changing your `/etc/docker/daemon.json` to look like this:

```
{
  "storage-driver": "overlay2",
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "5k",
    "max-file": "3",
    "compress": "true"
  }
}
```

2. Restart Docker so the new logging configuration takes effect:

```
[centos@ucp-node-0 ~]$ sudo service docker restart
```

3. Start another container generating logs:

```
[centos@ucp-node-0 ~]$ docker container run -d centos:7 ping 8.8.8.8
```

4. Find the container's log files under `/var/lib/docker/containers`:

```
[centos@ucp-node-0 ~]$ sudo ls -lsh /var/lib/docker/containers/<container ID>

<container ID>-json.log
<container ID>-json.log.1.gz
<container ID>-json.log.2.gz
...
```

At first you'll probably only see the `-json.log` file; the `.1.gz` and `.2.gz` will appear as log files get rotated out.

5. Keep listing the above directory every few seconds; you should see the original log file get rotated to `<container ID>-json.log.1.gz` once it reaches about 5 kb in size. Also, once it gets rotated out to `.1.gz`, it will be automatically compressed.

15.3 Conclusion

In this exercise, we reconfigured the Docker Engine's default logging options, and rotated out and compressed logfiles once they reached a certain size. An important aspect of cluster design is allocating and managing disk space for logs; while we can't provision an unlimited amount of disk on each of our nodes for logs, the more logs we're able to keep, the further back in history we can look when troubleshooting our deployments.

16 UCP Audit Logs

By the end of this exercise, you should be able to:

- Configure and retrieve UCP API audit logs via `ucp-controller` container logs

16.1 Configuring Audit Logs

1. Start by establishing API credentials so we can read and update audit log configuration. Recall that `<UCP FQDN>` is the FQDN of your ingress node if you're connecting to UCP through a load balancer there, or the FQDN of `ucp-manager-0` if you aren't load balancing:

```
[centos@infra ~]$ UCP_FQDN=<UCP FQDN>
[centos@infra ~]$ AUTHTOKEN=$(curl -sk \
    -d '{"username":"admin","password":"adminadmin"}' \
    https://${UCP_FQDN}/auth/login | jq -r .auth_token)
[centos@infra ~]$ alias ucp-api='curl -k -H "Authorization: Bearer $AUTHTOKEN"'
```

2. Fetch the current audit log configuration:

```
[centos@infra ~]$ ucp-api https://${UCP_FQDN}/api/ucp/config/logging \
    > auditlogconfig.json
[centos@infra ~]$ cat auditlogconfig.json

{"logLevel":"INFO","auditLevel":"","supportDumpIncludeAuditLogs":false}
```

3. In `auditlogconfig.json`, edit the `auditlevel` field to be the most verbose, `"auditlevel":"request"`.
4. Upload the audit log configuration back to UCP with a PUT request to the same endpoint:

```
[centos@infra ~]$ ucp-api -X PUT --data $(cat auditlogconfig.json) \
    https://${UCP_FQDN}/api/ucp/config/logging
```

Optional: do `ucp-api https://${UCP_FQDN}/api/ucp/config/logging` to read back the audit log config, and make sure your new settings got set successfully.

5. Log in to UCP as any user other than admin; I'll use `alice` in this example, but it can be anyone. Create any swarm service as user `alice`, for example with service name `demo` and image `nginx:latest`.
6. Search the logs of the `ucp-controller` container on `ucp-manager-0` for audit records from user `alice` hitting the `service/create` endpoint:

```
[centos@ucp-manager-0 ~]$ docker container logs \
    ucp-controller 2>&1 | grep "services/create.*alice" | jq
```

You should see a large JSON object describing the audit log, identifying the endpoint hit, the user who hit it, timestamps for the request, and other metadata describing the event. If the above command returns nothing, try it on `ucp-manager-1` and `ucp-manager-2`; the API call may have got load balanced to one of your other managers.

7. Clean up by deleting the service you just created.

16.2 Conclusion

In this exercise, we extracted audit logs from UCP, via the logs of the `ucp-controller` container. Audit logs are a key observability component that tracks all security-relevant events on the platform, and give complete historical perspectives on deployments to underwrite troubleshooting.

17 Centralized Logging

By the end of this exercise, you should be able to:

- Set up centralized logging for a swarm using an ELK stack

17.1 Installing ELK Stack

In this section we are going to install an ELK stack which serves as a log aggregator and offers a web frontend for the users to drill into logging data.

1. Make sure your `infra` node's Docker client is pointing at its own Docker engine (and not at the UCP cluster via the client bundle you were using earlier):

```
[centos@infra ~]$ unset DOCKER_TLS_VERIFY COMPOSE_TLS_VERSION \
    DOCKER_CERT_PATH DOCKER_HOST
```

2. Prepare `infra` to run `elasticsearch` by running the following commands:

```
[centos@infra ~]$ sudo sysctl -w vm.max_map_count=262144
[centos@infra ~]$ echo 'vm.max_map_count=262144' | sudo tee --append /etc/sysctl.conf
```

3. Clone a GitHub repository which contains the code necessary to define and run the ELK stack:

```
[centos@infra ~]$ git clone -b ee2.1 https://github.com/docker-training/elk-dee.git
[centos@infra ~]$ cd elk-dee
```

4. We want our logging stack to be completely independent from our cluster, so that if the cluster goes down, it doesn't take our logging infrastructure with it. As such, make the `infra` node its own swarm:

```
[centos@infra elk-dee]$ docker swarm init
```

5. Finally we deploy the ELK stack:

```
[centos@infra elk-dee]$ docker stack deploy -c elk-docker-compose.yml elk
```

Double check that everything runs by using this command:

```
[centos@infra elk-dee]$ watch docker service ls
```

and wait until every service is running. You should see something like this:

```
Every 2.0s: docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
13sksh79tim2	elk_elasticsearch	replicated	1/1	elasticsearch:5.2
cluyclmtaw48	elk_logstash	replicated	1/1	logstash:5.2-alpine
whni0r8ddm8i	elk_kibana	replicated	1/1	kibana:5.2

specifically note the column **Replicas** saying 1/1 for every service. This indicates that we're ready to roll.

17.2 Configuring all Swarm nodes

Now we need to configure the swarm so that every node in it reports all its logs to the ELK stack. We'll use the `journald` log driver, so we can forward container and system logs to the ELK stack, all from the system journal.

1. SSH into the `ucp-manager-0` node.
2. Add this logging configuration to the file `/etc/docker/daemon.json`, alongside the storage driver configuration. The final file should look like:

```
{
  "storage-driver": "overlay2",
  "log-driver": "journald",
  "log-level": "error",
  "log-opts": {
    "tag": "{{.ImageName}}/{{.Name}}/{{.ID}}"
  }
}
```

3. Restart the Docker daemon:

```
[centos@ucp-manager-0 ~]$ sudo service docker restart
```

4. Repeat the above steps **every single** node of the swarm.

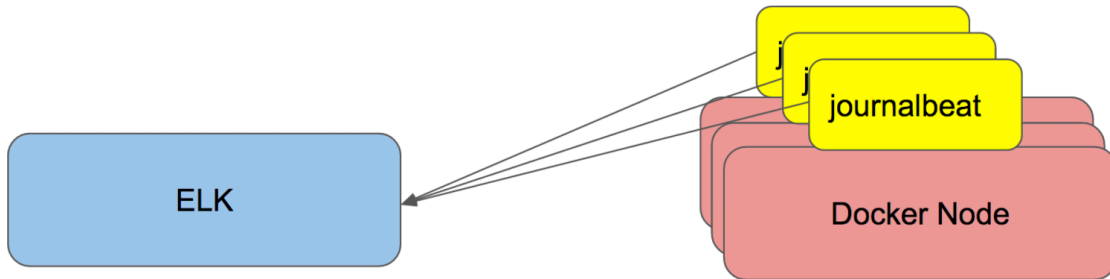
17.3 Stream all Docker logs to ELK

Now that we have configured all nodes to generate their respective logs using the `journald` driver we need to stream the data to our ELK stack.

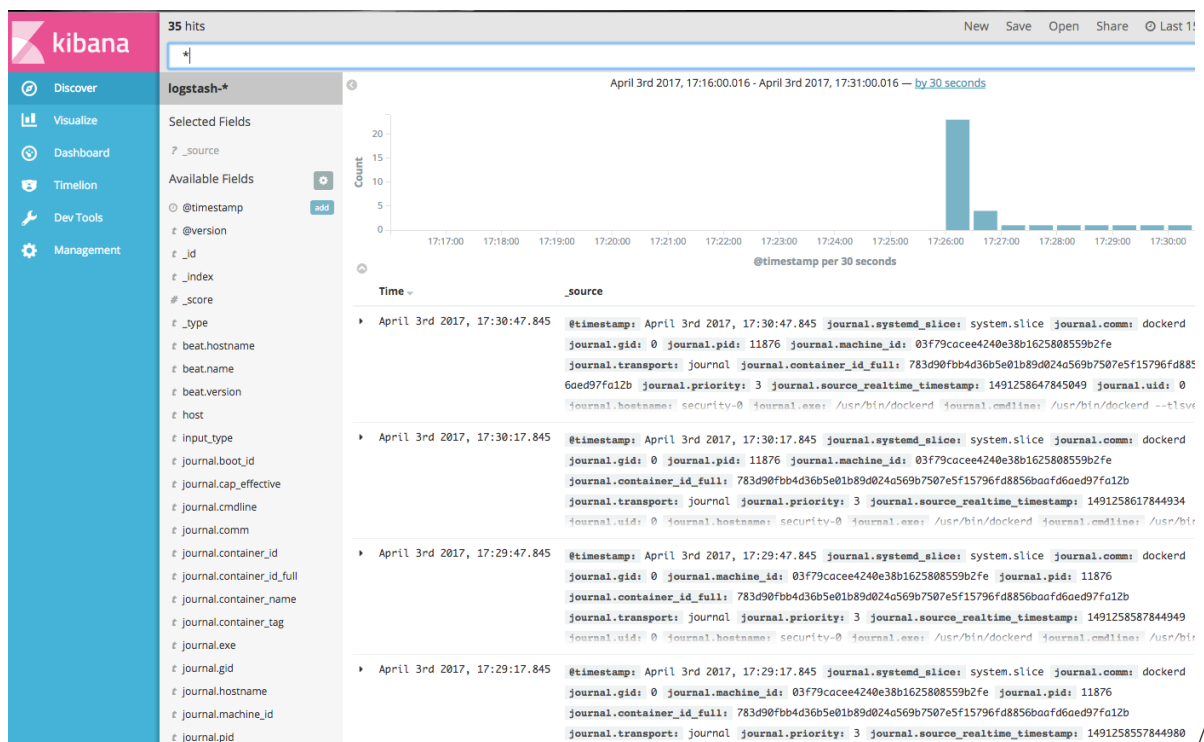
1. SSH into `ucp-manager-0`, and run the following:

```
[centos@ucp-manager-0 ~]$ git clone -b ee2.1 \
  https://github.com/docker-training/elk-dee.git
[centos@ucp-manager-0 ~]$ cd elk-dee
[centos@ucp-manager-0 elk-dee]$ export LOGSTASH_HOST=<infra private IP>
[centos@ucp-manager-0 elk-dee]$ docker stack deploy \
  -c journalbeat-docker-compose.yml journalbeat
```

The result of this can be visualized as follows:



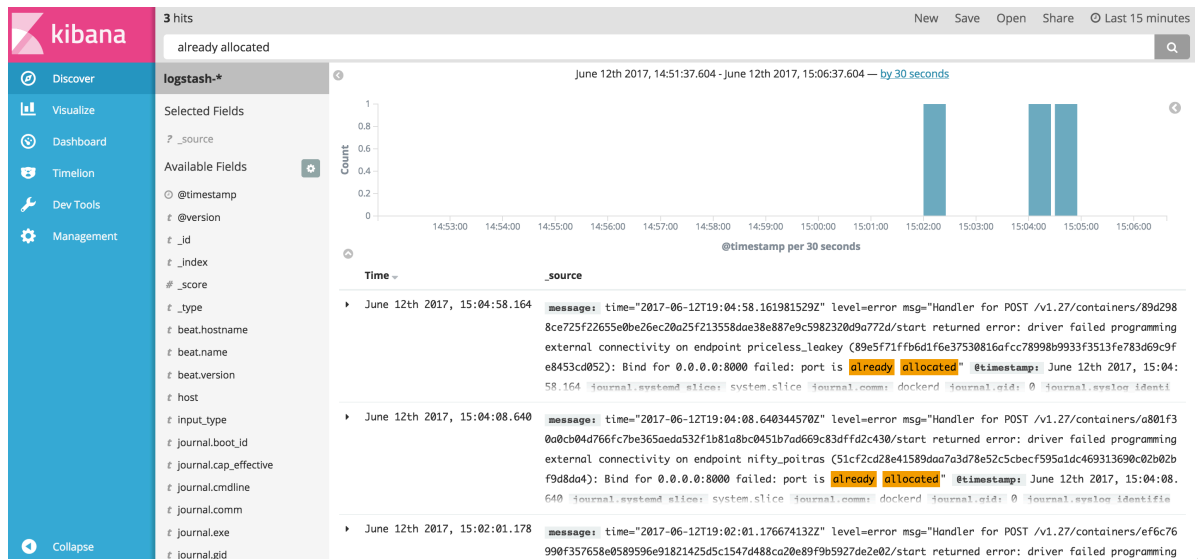
2. Open a browser at `http://<infra public IP>:5601` to access Kibana. Upon first usage you should be asked to create an index for the events before you can proceed. Accept the defaults and click **Create**.
3. In Kibana navigate to **Discover** and view the list of captured events.



4. If you don't see any events yet, try to launch two containers listening on the same port on your UCP cluster, to generate an error:

```
[centos@ucp-manager-0 ~]$ docker container run -d -p 8000:80 nginx
[centos@ucp-manager-0 ~]$ docker container run -d -p 8000:80 nginx
```

5. In the filter box at the top of the Kibana page enter `*already allocated` and hit enter. Port collision errors should be reported and plotted:



17.4 Conclusion

In this exercise, you set up a basic ELK stack. ELK is just one of many options for ingesting and collating Docker's logs, but they all rely on the logging configuration established by `/etc/docker/daemon.json`. For more information on log drivers in Docker, see <https://docs.docker.com/engine/admin/logging/overview/>.

18 Health Checks

By the end of this exercise, you should be able to:

- Specify a healthcheck endpoint in a Dockerfile or a yaml description of a Kubernetes deployment
- Configure healthcheck probes in Swarm and Kubernetes

18.1 Analyzing the Dockerfile

- Please visit the source code to this service, which can be found here: <https://github.com/docker-training/healthcheck.git>.
- First let's look at the Dockerfile for this service. It looks as follows:

```
FROM ubuntu:16.04

RUN apt-get update && apt-get -y upgrade
RUN apt-get -y install python-pip curl
RUN pip install flask==0.10.1

ADD /app.py /app/app.py
WORKDIR /app

HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit 1

CMD python app.py
```

- Please specifically note the `HEALTHCHECK` line, which defines the command used to evaluate the health of the application. Exit code 0 is interpreted as healthy, and exit code 1 is interpreted as unhealthy.
- Have a look at the application code itself; the most important part for healthchecking is the `/health` route:

```
@app.route('/health')
def health():
    global healthy

    if healthy:
        return 'OK', 200
    else:
        return 'NOT OK', 500
```

The app performs some logic to decide if it is healthy or not when `/health` is visited. This toy example just checks a bit, but a real example would have the same structure.

18.2 Deploying a Healthcheck-Enabled Service

1. Deploy a service with healthchecks enabled. This service will perform a healthcheck every two seconds; wait two seconds for a healthy response each time; declare a container failed after three consecutive failures; and wait 10 seconds after container launch to begin the healthchecks:

```
[centos@ucp-manager-0 ~]$ docker service create --name app \
  --health-interval 2s \
  --health-timeout 2s \
  --health-retries 3 \
  --health-start-period 10s \
  -p 5000:5000 training/healthcheck:ee2.1
```

2. Open a second SSH connection to `ucp-manager-0` and run the following command to observe the app service:

```
[centos@ucp-manager-0 ~]$ watch docker service ps app
```

you should see something like this:

```
Every 2.0s: docker service ps app
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE
ralw1apn8mgs	app.1	training/healthcheck:ee2.1	ucp-node-0	Running	Running 25 seconds

As we can see, the service is up and running happily.

3. But now we want to disrupt this peaceful state a bit. In your first terminal execute the command to put the service into an unhealthy status:

```
[centos@ucp-manager-0 ~]$ curl localhost:5000/kill -4
```

This flips the health bit in our app, so it starts reporting as unhealthy.

4. Observe what's happening in the second terminal where you run the `watch` command. After approximately 8 seconds you should see that the running instance gets killed and a new instance is started instead.
5. Explain the delay between the kill command and the moment the service gets effectively killed.
6. If Kibana is still running from a previous exercise, visit your Kibana UI and search for `unhealthy` container to see the system events logged when a healthcheck killed the container.
7. To get some detailed information about the last five healthchecks on a container, find the node the container is running on and do:

```
[centos@ucp-x ~]$ docker container inspect \
  --format '{{json .State.Health.Log}}' <container id> | jq
```

8. Clean up the system by removing the service:

```
[centos@ucp-manager-0 ~]$ docker service rm app
```


18.4 Conclusion

In this exercise, we stepped through the configuration and behavior of healthcheck-enabled containers in both Swarm and Kubernetes. All Swarm services and Kubernetes deployments should have healthchecks defined for them, as part of the automation and self-healing of your containerized workloads. Make sure your developers understand this requirement, so they can provide insightful healthcheck endpoints in their containerized software.

19 Installing Docker Trusted Registry

By the end of this exercise, you should be able to:

- Install a single replica of Docker Trusted Registry on top of an existing UCP deployment

Pre-requisites:

- UCP installed with at least one manager node

19.1 Add additional worker nodes to UCP

DTR is installed on top of UCP nodes. In a real deployment, we would add worker nodes dedicated to running DTR, and install DTR there. However for this demo, we'll just run DTR on the same nodes as our UCP managers, so no extra nodes need to be added at this step. **Do not do this in production!** Always provision dedicated nodes for DTR in a real environment.

19.2 Installing DTR

1. Navigate **admin** -> **Admin Settings** -> **Scheduler**.
2. Tick the box that says **Allow administrators to deploy containers on UCP managers or nodes running DTR**. We need to enable this in order to be able to install DTR on our UCP nodes. Make sure to save this config if necessary.
3. By default, DTR uses ports 80 and 443 on whichever host it's installed on, but in our case, we're going to customize this to 81 and 4443 instead, since we'll be installing DTR on the same nodes as our UCP managers. Make sure nothing is listening on tcp/81 or tcp/4443 before continuing.
4. SSH into `ucp-manager-0`, and run the following commands. Note that in our case `DTR_FQDN` is going to be set to the FQDN of `ucp-manager-0`, since we're running DTR on the same nodes as UCP managers.

```
[centos@ucp-manager-0 ~]$ DTR_FQDN=<ucp-manager-0 FQDN>
[centos@ucp-manager-0 ~]$ UCP_IP=<ucp-manager-0 public IP>
[centos@ucp-manager-0 ~]$ docker container run -it --rm docker/dtr:2.6.0 install \
  --ucp-node ucp-manager-0 \
  --ucp-username admin \
  --ucp-password adminadmin \
  --ucp-url https://${UCP_IP} \
  --ucp-insecure-tls \
  --replica-https-port 4443 \
  --replica-http-port 81 \
  --dtr-external-url https://${DTR_FQDN}:4443
```

Note the `--ucp-password` flag above; if you changed your admin password when you installed UCP or during the Password Recovery exercise, you'll need to use that password here instead of `adminadmin`.

5. Wait for the install to complete, then navigate to **Shared Resources** -> **Stacks** on the UCP left sidebar. DTR should be listed there, with 9 services.
6. Open a new browser tab and navigate to `https://<ucp-manager-0 FQDN>:4443`; after accepting another security exception, and logging in with your UCP admin credentials, the DTR dashboard is presented.

19.3 Conclusion

In this exercise we have installed Docker Trusted Registry, which is now running on top of UCP. Note that DTR installation expects ports tcp/443 and tcp/80 to be available by default (4443 and 81 in our example above); if they aren't, double check whether one of those ports are occupied by a service in UCP (including the HTTP routing mesh, which defaults to tcp/80). At this point, we have a minimal viable installation; in the next few exercises, we'll learn how to set DTR in high availability mode, provide storage backings for images, configure load balancers, and authenticate a remote Docker Engine with our new registry.

20 (Optional) Configuring DTR for High Availability

By the end of this exercise, you should be able to:

- Configure a custom DTR storage backend
- Add a DTR replica to an existing deployment so DTR is highly available (HA).
- Configure a load balancer and certificates correctly for directing traffic to an HA DTR.

Pre-requisites:

- UCP and DTR set up as described in the 'Installing DTR' exercise.

Note: in this exercise, we'll continue setting up DTR replicas on our UCP managers. As noted in the last exercise, this is fine for a demo, but should not be done in a business critical environment! In practice, DTR should be installed on dedicated UCP workers.

20.1 Setting up a DTR Storage Backend

By default, all images uploaded to DTR will be kept on the local disk of the first DTR replica you just set up. If this node goes down, you will lose access to your images, *even if you have extra DTR replicas still working*. In order to provide true high availability, we must also set up a highly available storage backend for DTR. In this example, we'll deploy minio, a containerized, highly available database with an S3-compliant API. This minio deployment is for demonstration purposes only! In production, feel free to pick any image storage solution you're comfortable with from the options discussed in the docs, but do **not** fail to pick one and set it up. Otherwise, your images will not be highly available.

1. We'll run minio as a Swarm stack on our `infra` node. Start by making sure your `infra` node's Docker client is pointing at its own Docker engine (and not at the UCP cluster via the client bundle you were using earlier):

```
[centos@infra ~]$ unset DOCKER_TLS_VERIFY COMPOSE_TLS_VERSION \
  DOCKER_CERT_PATH DOCKER_HOST
```

2. Put `infra` into swarm mode if it isn't already, by doing `docker swarm init` there.

Note we are *not* adding `infra` to our existing swarm! We want our DTR storage backing to remain independent from our cluster, so if we lose one, we don't lose both.

3. On `infra`, fetch the minio demo setup script, source it, and wait for your minio services to report up and running:

```
[centos@infra ~]$ wget https://bit.ly/2SujSvt -O minio-demo.sh
[centos@infra ~]$ source minio-demo.sh
[centos@infra ~]$ docker stack ps minio_stack

... NAME                ... NODE ... CURRENT STATE ...
... minio_stack_minio4.1 ... infra ... Running 6 seconds ago ...
... minio_stack_minio3.1 ... infra ... Running 6 seconds ago ...
... minio_stack_minio2.1 ... infra ... Running 5 seconds ago ...
... minio_stack_minio1.1 ... infra ... Running 6 seconds ago ...
```

Minio takes about 30 seconds to start up; wait until everything reports running, like above.

4. Visit minio at <infra public IP>:9001. Login with access and secret keys password and password.
5. Click the red '+' in the bottom right of minio, and then click 'Create Bucket'. Name your bucket dtr.
6. In DTR as an administrator, navigate **System** -> **Storage**, then click **Cloud** -> **Amazon S3** to set up an S3-compliant database. Note that despite what the button says, we *aren't* going to use an S3 bucket; any database with an S3-compliant API, such as minio, can be set up as a storage backing from here.
7. In the **S3 Settings** form, fill out the following fields (leave the rest blank):
 - **AWS Region Name:** us-east-1 (no matter where your nodes are - always us-east-1 for minio).
 - **S3 Bucket Name:** dtr
 - **Region Endpoint:** http://<infra public IP>:9001
 - **AWS Access Key:** password
 - **AWS Secret Key:** password

Click **Save**. DTR should report successfully saving the settings.

20.2 Installing Replicas

1. On your ucp-manager-0 node, run the docker/dtr bootstrapper to configure ucp-manager-1 as a DTR replica:

```
[centos@ucp-manager-0 ~]$ UCP_FQDN=<ucp-manager-0 FQDN>
[centos@ucp-manager-0 ~]$ docker container run -it --rm docker/dtr:2.6.0 join \
  --ucp-node ucp-manager-1 \
  --ucp-username admin \
  --ucp-password adminadmin \
  --ucp-url https://{UCP_FQDN} \
  --ucp-insecure-tls \
  --replica-https-port 4443 \
  --replica-http-port 81
```

You'll be asked to 'Choose a replica to join'; press return to accept the default (the only currently available option). Take note of this replica ID. **Note** the --ucp-password flag above; if you changed your admin password when you installed UCP or during the Password Recovery exercise, you'll need to use that password here instead of adminadmin.

2. Run the bootstrapper again but this time, change the --ucp-node flag to ucp-manager-2. When asked which replica to join, make sure to choose the same one you did in the last step (might not be the default this time).
3. Check the **Stacks** page inside the UCP web UI. You should now see three instances of DTR.

TYPE	NAME	SERVICES/CONTAINERS	NETWORKS	VOLUMES	SECRETS
Basic Conta...	Docker Trusted Registry 2.6.0-tp8 - (Replica 2b9f5eb96765)	9	0	0	-
Basic Conta...	Docker Trusted Registry 2.6.0-tp8 - (Replica 500a3fb5ceb9)	9	0	0	-
Basic Conta...	Docker Trusted Registry 2.6.0-tp8 - (Replica d7595de84476)	9	0	0	-
Basic Conta...	Docker Universal Control Plane h5n7wiltfourwhrs3xr5pohz	6	0	0	-
Swarm Serv...	journalbeat	1	1	0	0

20.3 Load Balancing DTR

Now that we have three DTR replicas, we'd like to load balance requests across them. We'll configure an nginx load balancer on your infra node for this purpose.

1. On your infra node, create a file called `dtr-nginx.conf`, and place the following content in it; make sure to replace the `<variables>` with the public IPs of your own manager nodes:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

stream {
    upstream dtr_4443 {
        server <ucp-manager-0 public IP>:4443;
        server <ucp-manager-1 public IP>:4443;
        server <ucp-manager-2 public IP>:4443;
    }
    server {
        listen 4443;
        proxy_pass dtr_4443;
    }
}
```

2. Deploy a containerized nginx server, mounting the config you just created:

```
[centos@infra ~]$ docker container run -d --name dtr-lb --restart=unless-stopped \
  --publish 4443:4443 \
  --volume ${PWD}/dtr-nginx.conf:/etc/nginx/nginx.conf:ro \
  nginx:stable-alpine
```

3. Visit DTR at `https://<ucp-manager-0 FQDN>:4443` (note *not* the load balancer address yet), and then:

- Navigate **System -> General**
- Scroll down to *Domain & Proxies*
- In the field **Load Balancer / Public Address**, enter your load balancer's FQDN and port, `<infra FQDN>:4443`, and click **Save**.
- Navigate in your browser to `https://<infra FQDN>:4443`. You should now be able to successfully reach DTR through your load balancer.

Note: you have now successfully placed a load balancer in front of your DTR deployment. When connecting to DTR in future exercises, `<DTR_FQDN>` will be the FQDN of your infra node. DTR will only accept connections from the public address you registered under **System -> General Domain & Proxies** above.

20.4 Cleanup

1. Navigate **Admin Settings -> Scheduler** in UCP
2. Untick both checkboxes under **Container Scheduling**. This is so that later on, when we run containers, we do not accidentally get them scheduled on our DTR nodes or on our UCP manager nodes.

20.5 Conclusion

After completing this exercise, your DTR instance is running in high availability mode. Just like the UCP manager consensus, the DTR replicas maintain their state in a database, and elect a leader via a Raft consensus.

21 Pushing and Pulling From DTR

By the end of this exercise, you should be able to:

- Establish certificate trust between any Docker Engine and DTR
- Push and pull images from DTR

21.1 Integrate UCP and DTR

By default Docker engine uses TLS when pushing and pulling images to an image registry like Docker Trusted Registry; as such, each Docker engine must be configured to trust DTR.

1. Switch into your `ucp-manager-0` terminal. In `~/.bashrc`, add the following line at the bottom:

```
export DTR_FQDN=<DTR FQDN>
```

Where `<DTR FQDN>` is the FQDN you plan to access DTR at. This depends on how you set up DTR:

- **If you skipped** setting up a load balancer for DTR, then use the FQDN for `ucp-manager-0`, your single DTR replica.
- **If you have a load balancer for DTR**, then use the FQDN for the load balancer host, which should be that of your infra node.

Keep track of this DTR FQDN. We will use it throughout the following exercises; anywhere we refer to `<DTR FQDN>`, this is the FQDN we mean.

Source this file via `source ~/.bashrc` so these changes take effect in the current terminal.

2. Download the DTR certificate:

```
[centos@ucp-manager-0 ~]$ sudo curl -k https://${DTR_FQDN}:4443/ca \
-o /etc/pki/ca-trust/source/anchors/${DTR_FQDN}:4443.crt
```

3. Refresh the list of certificates to trust:

```
[centos@ucp-manager-0 ~]$ sudo update-ca-trust
```

4. Restart the Docker daemon:

```
[centos@ucp-manager-0 ~]$ sudo /bin/systemctl restart docker.service
```

5. Log in to DTR from the command line:

```
[centos@ucp-manager-0 ~]$ docker login ${DTR_FQDN}:4443
```

If your login is successful, you have configured your Docker engine to trust DTR.

6. Repeat steps 1-5 on all your VMs (use the script below); every Docker engine in the Swarm cluster must be configured to trust DTR.

Note: To speed up the whole process of configuring all nodes of our swarm we can use a bash script like the following:

```
CERT=./<your-pem-name>.pem
DTR_FQDN=<DTR FQDN>
NODE_LIST=<PUBLIC-IP-ADDRESSES-OF-NODES-IN-SWARM>
# e.g. ("52.23.162.247" "52.3.334.184" "52.205.102.106" ...)
for NODE in "${NODE_LIST[@]"; do
```

```
ssh-keyscan -H $NODE >> ~/.ssh/known_hosts
ssh -t -i $CERT centos@$NODE hostname
ssh -t -i $CERT centos@$NODE sudo curl -k https://${DTR_FQDN}:4443/ca \
-o /etc/pki/ca-trust/source/anchors/${DTR_FQDN}:4443.crt
ssh -t -i $CERT centos@$NODE sudo update-ca-trust
ssh -t -i $CERT centos@$NODE sudo /bin/systemctl restart docker.service
done;
```

Replace the placeholders with the respective values of your infrastructure.

21.2 Pushing your first image

1. Via the DTR web UI, and click on the **New Repository** button to create a repository called `hello-world`. The repository should go under your *admin* account.

2. On `ucp-manager-0`, pull the `hello-world` image from Docker Hub:

```
[centos@ucp-manager-0 ~]$ docker image pull hello-world:latest
```

3. Re-tag the image, and note the correct namespacing for pushing to your DTR; the top level namespace must be the DTR FQDN and external port:

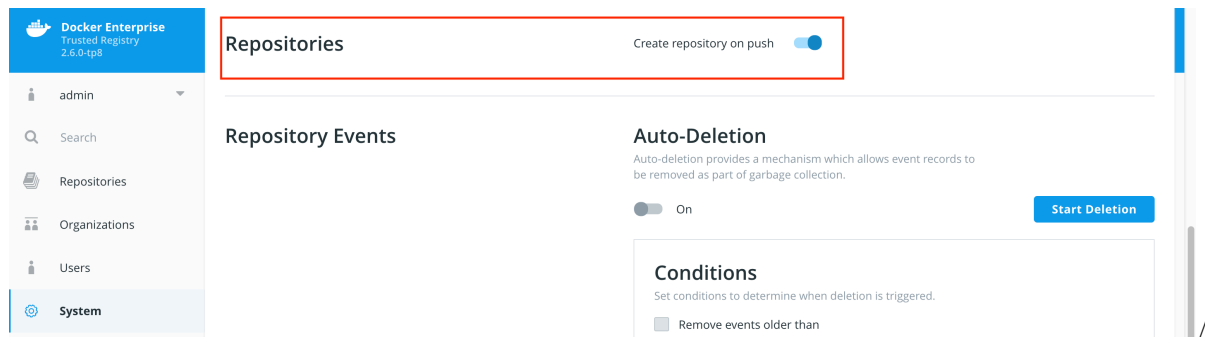
```
[centos@ucp-manager-0 ~]$ docker image tag hello-world:latest \
${DTR_FQDN}:4443/admin/hello-world:1.0
```

4. Push to DTR:

```
[centos@ucp-manager-0 ~]$ docker image push ${DTR_FQDN}:4443/admin/hello-world:1.0
```

and verify that you can see your image in DTR by navigating **Repositories** -> **admin/hello-world** -> **Tags**.

5. Now, we will configure the option to auto create a repository on push. In the DTR web UI, navigate **System** -> **General**, and enable **Create repository on push**:



6. Back on `ucp-manager-0`, re-tag the `hello-world` image for a repository that has not yet been created:

```
[centos@ucp-manager-0 ~]$ docker image tag \
hello-world:latest ${DTR_FQDN}:4443/admin/hellohello
```

7. Push to DTR:

```
[centos@ucp-manager-0 ~]$ docker image push ${DTR_FQDN}:4443/admin/hellohello
```

and verify that a new repository has been created and that you can see your image.

21.3 Conclusion

In this exercise, we completed setting up DTR by establishing certificate trust between DTR and any Docker Engine that wants to interact with it. Also note the example of correct image namespacing: image names must begin with

the trusted host for pushing to DTR, in this case `${DTR_FQDN}:4443/`. Failure to namespace images correctly will result in rejected attempts to `docker image push`.

22 Working with Organizations and Teams

By the end of this exercise, you should be able to:

- Grant read and/or write access to a DTR repository based on user team membership
- Create a DTR access token to allow a user to make authenticated pushes and pulls from DTR without risking exposing their password

Pre-requisites:

- DTR set up as per the 'Install DTR' exercise above.

22.1 Creating Organizations and Teams

1. Create two new user accounts, for users *chandrasekhar* and *suzuki*. You must be logged in as an admin to do this.
2. Create a new organization called **myorg** in DTR (navigate **Organizations** -> **New Organization**).
3. Create two teams under myorg, **database** and **web** (navigate **Organizations** -> **myorg** -> **Teams** +).
4. Add Chandrasekhar to the database team, and Suzuki to the web team (navigate **Organizations** -> **myorg** -> **database** -> **Add User**).
5. Create a repository **myorg/db** owned by myorg: navigate **Organizations** -> **myorg** -> **Repositories** -> **New Repository**. Do the same again for another repo **myorg/ui**.
6. Give the database team R/W access to **myorg/db**: navigate **Organizations** -> **myorg** -> **database** -> **Repositories** -> **New Repository** -> **Existing**, and pick the appropriate choices from the dropdown lists.
7. Finally, assign the following permissions in the same fashion:
 - database team, R/O access to **myorg/ui**
 - web team, R/W access to **myorg/ui**
 - web team, R/O access to **myorg/db**

22.2 R/W in Org Repos

1. Log in to the DTR web UI as **Suzuki**, and click on Repositories on the left sidebar to see a list of all repos you have read access to. Use the dropdown to filter only repos that belong to the myorg org.
2. On `ucp-manager-0`, tag any arbitrary image as `${DTR_FQDN}:4443/myorg/ui:1.0`
3. Log in to DTR from the bash shell as Suzuki, and push the ui image you just tagged. Everything should proceed as normal.

```
[centos@ucp-manager-0 ~]$ docker login ${DTR_FQDN}:4443
```

4. Tag another image as `${DTR_FQDN}:4443/myorg/ui:1.1`, log in to DTR from the shell as Chandrasekhar this time, and push the image. The image push should fail - why?
5. Log into the DTR web UI as admin, and set both the db and ui repositories to **Private** (look under the **Settings** tab for each). A 'private' icon should appear next to the repo name when this has been set correctly. Then log in to the DTR web UI as Suzuki and check if that user can still see myorg/ui and myorg/db. Why or why not?
6. Finally, revoke the *database* team's read only permission to the ui repo, and the *web* team's read only permission to the db repo. Can Suzuki see both repositories now? Why or why not?

22.3 Establishing Access Tokens

1. Log into the DTR web UI as admin, and navigate **Users -> Suzuki -> Access Tokens**.
2. Click **New access token**, give the token a name, and click **Create**. An access token is presented *only this once!* Copy it, don't lose it, and click **Done**.
3. Try logging into DTR from the CLI as Suzuki, using this new access token instead of Suzuki's usual password. It should succeed.
4. Try logging into the DTR or UCP web frontend with this token; the login will fail. Access tokens are only for docker login purposes - not authenticating with the EE API.

22.4 Conclusion

In this exercise, we saw how organization-owned repositories appear on DTR. Users can read and write to repos based on the permissions afforded them by their team; private org-owned repos become invisible to any user without at least explicit read access to that repo.

Access tokens can also be granted to users to obfuscate upstream passwords; if an access token is compromised, it can be rotated out without revealing the user's root password.

23 Content Trust

A key responsibility of an operations team is to ensure only approved software makes it into production. In this exercise, you'll learn how to use Docker Content Trust to allow users to sign Docker images, and how to set UCP to only run images with a valid set of signatures.

By the end of this exercise, you should be able to:

- Set up content trust keys using `docker trust` commands
- Generate content trust metadata for a repository and tag in DTR
- Determine which tags in a repository have trust metadata
- Grant and revoke signing authority for a DTR repository to a DTR user
- Revoke content trust metadata
- Enforce content trust signatures on all running containers in UCP
- Enforce trust pinning on images pulled and run by an individual Docker engine

As always, make sure you complete each step successfully before moving onto the next, and ask for help if something is unclear. This is especially important with an enforcement mechanism like content trust; it will fail *on purpose* if anything looks amiss.

23.1 Setup

Before we begin establishing content trust, set up a few resources in DTR:

- Create an organization called *demo* (if you already have such an organization, go ahead and use that one)
- Add a team *QA* to the *demo* organization
- Create a user *abe*, and add them to the *QA* team
- Add a repository *example* to the *demo* organization
- Give the *QA* team write access to the *example* repository

23.2 Establishing Trust

When we first begin signing images for a given repository, a few keys need to be generated and initial trust metadata established; in this section, we'll get started by setting a DTR administrator up to sign trust metadata for a single repository.

1. On ucp-node-0, pull, tag, and push an image to your new repository exactly as you did in previous exercises:

```
[centos@ucp-node-0 ~]$ docker image pull alpine:latest
[centos@ucp-node-0 ~]$ DTR_FQDN=<Public DNS where you are reaching DTR>
[centos@ucp-node-0 ~]$ docker image tag \
    alpine:latest ${DTR_FQDN}:4443/demo/example:untrusted
[centos@ucp-node-0 ~]$ docker login ${DTR_FQDN}:4443 -u admin
[centos@ucp-node-0 ~]$ docker image push ${DTR_FQDN}:4443/demo/example:untrusted
```

Check the DTR web frontend to make sure your :untrusted tag is present.

2. Next we'll set up the admin user's trust keys and sign and push trust metadata for a new tag in this repository. Start by making a new tag:

```
[centos@ucp-node-0 ~]$ docker image tag \
    alpine:latest ${DTR_FQDN}:4443/demo/example:trusted
```

Now sign and push this image using the docker trust CLI. You'll be asked to generate a number of passwords; keep track of these and what keys they belong to:

```
[centos@ucp-node-0 ~]$ docker trust sign ${DTR_FQDN}:4443/demo/example:trusted
```

You are about to create a new root signing key passphrase. This passphrase will be used to protect the most sensitive key in your signing system. Please choose a long, complex passphrase and be careful to keep the password and the key file itself secure and backed up. It is highly recommended that you use a password manager to generate the passphrase and keep it safe. There will be no way to recover this key. You can find the key in your config directory.

Enter passphrase for new root key with ID 2f602d2:

Repeat passphrase for new root key with ID 2f602d2:

Enter passphrase for new repository key with ID dad1fba:

Repeat passphrase for new repository key with ID dad1fba:

Enter passphrase for new admin key with ID 00729ba:

Repeat passphrase for new admin key with ID 00729ba:

Created signer: admin

Finished initializing signed repository for ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example

Signing and pushing trust data for local image ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example

The push refers to repository [ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example]

73046094a9b8: Layer already exists

trusted: digest: sha256:0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b size: 528

Signing and pushing trust metadata

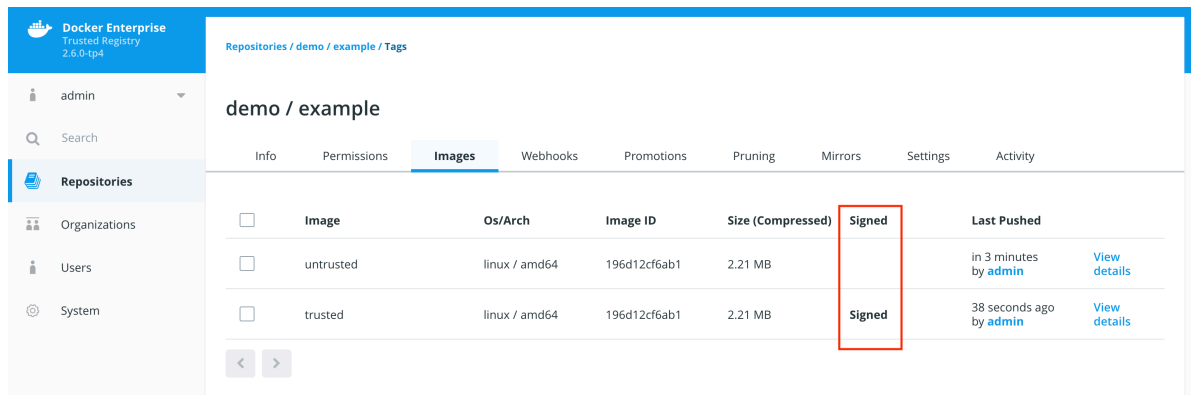
Enter passphrase for admin key with ID 00729ba:

Successfully signed ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example:trusted

Keep track of the root key ID (2f602d2 in the above example); we'll need it for another step at the end of this exercise.

At this point, you have generated the root content trust key, a repository key for \${DTR_FQDN}:4443/demo/example:trusted, and a signing key for the admin user; all of these keys are stored under /home/centos/.docker/trust/private.

Check your DTR web frontend; you should see your :trusted tag with a 'Signed' badge, indicating it carries a content trust signature:



3. On ucp-node-1, set Docker into content trust enforcing mode:

```
[centos@ucp-node-1 ~]$ export DOCKER_CONTENT_TRUST=1
```

Login as user Abe, and attempt to pull the :untrusted tag:

```
[centos@ucp-node-1 ~]$ DTR_FQDN=<Public DNS where you are reaching DTR>
[centos@ucp-node-1 ~]$ docker login ${DTR_FQDN}:4443 -u abe
[centos@ucp-node-1 ~]$ docker image pull ${DTR_FQDN}:4443/demo/example:untrusted
```

No valid trust data for untrusted

Docker has declined to pull the requested image; why do you think that is?

4. Determine which tags in the demo/example repository have trust metadata by looking at the SignedTags list in the output of `docker trust inspect` for this repository:

```
[centos@ucp-node-1 ~]$ docker trust inspect ${DTR_FQDN}:4443/demo/example
[
  {
    "Name": "ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example",
    "SignedTags": [
      {
        "SignedTag": "trusted",
        "Digest": "0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b",
        "Signers": [
          "admin"
        ]
      }
    ]
  },
  ...
]
```

From the SignedTags list, we can see the only tag with a signature is :trusted. Pull this tag:

```
[centos@ucp-node-1 ~]$ docker image pull ${DTR_FQDN}:4443/demo/example:trusted
```

```
Pull (1 of 1): ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example:trusted@sha256:0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b: Pulling from demo/example
8e3ba11ec2a2: Already exists
Digest: sha256:0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b
Status: Downloaded newer image for ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example@sha256:0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b
Tagging ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example@sha256:0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b
```

Notice that the image is actually getting pulled by sha256 - the sha256 that was listed in the Digest key of the SignedTags block above. Only after the pull completes is the image re-tagged as `${DTR_FQDN}:4443/demo/example:trusted`.

23.3 Granting Signing Authority

We will often want to allow a number of different users to sign images as trusted. In this section, we'll grant signing authority to user Abe for our demo/example repository.

1. Look at the output of `docker trust inspect` above. Under the Signers list, you can see the only DTR user with current signing authority for this repository is user *admin*.
2. In order to grant signing authority, we'll need access to Abe's client bundle. Fetch it on ucp-node-0:

```
[centos@ucp-node-0 ~]$ UCP_FQDN=<Public DNS UCP is reachable at>
[centos@ucp-node-0 ~]$ AUTHTOKEN=$(curl -sk \
    -d '{"username":"abe","password":"<abe's password>"}' \
    https://${UCP_FQDN}/auth/login | jq -r .auth_token)
[centos@ucp-node-0 ~]$ alias ucp-api='curl -k -H "Authorization: Bearer $AUTHTOKEN"'
[centos@ucp-node-0 ~]$ ucp-api https://${UCP_FQDN}/api/clientbundle -o bundle.zip
[centos@ucp-node-0 ~]$ unzip bundle.zip ; ls

bundle.zip  ca.pem  cert.pem  cert.pub  env.cmd  env.ps1  env.sh  key.pem  kube.yml
```

The administrator needs that `cert.pem` public key certificate in order to grant Abe signing access.

3. Establish Abe as a signer for the demo/example repository. Updating signing authority for a repository requires the password for the repository key you generated above:

```
[centos@ucp-node-0 ~]$ docker trust signer \
    add --key cert.pem abe ${DTR_FQDN}:4443/demo/example
```

Adding signer "abe" to ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example...

Enter passphrase for repository key with ID dad1fba:

Successfully added signer: abe to ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example

4. Re-list the trust metadata summary for this repository, and look for the Signers list in the output:

```
[centos@ucp-node-0 ~]$ docker trust inspect ${DTR_FQDN}:4443/demo/example

...
  "Signers": [
    {
      "Name": "admin",
      "Keys": [
        {
          "ID": "00729ba49a8a74053d892b1ac68b7d2ef96175f032879a568438ba63b4a6ee83"
        }
      ]
    },
    {
      "Name": "abe",
      "Keys": [
        {
          "ID": "523c230738e0995ada35ee8d4d1aa2acf7f289467aa6c535605a5c39e7e8105c"
        }
      ]
    }
  ],
  ...
```

We can see that Abe has been added to the Signers list.

5. At this point, Abe is allowed to sign images using the private key that corresponds to the public certificate we registered as a valid signer in the last step. In order to actually sign images, Abe needs to load that private key

into Docker on the machine they're signing and pushing from. On ucp-node-1, copy Abe's client bundle over:

```
[centos@ucp-node-1 ~]$ scp centos@<ucp-node-0 public ip>:bundle.zip .
[centos@ucp-node-1 ~]$ unzip bundle.zip
```

6. Load the private key found in key.pem into Docker's trust infrastructure:

```
[centos@ucp-node-1 ~]$ sudo chmod 400 key.pem
[centos@ucp-node-1 ~]$ docker trust key load key.pem
```

```
Loading key from "key.pem"...
Enter passphrase for new signer key with ID 523c230:
Repeat passphrase for new signer key with ID 523c230:
Successfully imported key from key.pem
```

Here Abe establishes a password for their signing key (523c230 in my example) - note this matches the key identity shown for Abe in the output of `docker trust inspect` above.

7. Make sure you are logged in as Abe by running `docker login ${DTR_FQDN}:4443` on ucp-node-1 again.
8. Tag, sign and push an image as Abe:

```
[centos@ucp-node-1 ~]$ docker image pull centos:7
[centos@ucp-node-1 ~]$ docker image tag \
    centos:7 ${DTR_FQDN}:4443/demo/example:trust-centos
[centos@ucp-node-1 ~]$ docker trust sign ${DTR_FQDN}:4443/demo/example:trust-centos
```

```
Signing and pushing trust data for local image ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/
The push refers to repository [ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example]
1d31b5806ba4: Pushed
trust-centos: digest: sha256:fc2476ccae2a5186313f2d1dad4a969d6d2d4c6b23fa98b6c7b0a1faad67685 size:
Signing and pushing trust metadata
Enter passphrase for signer key with ID 523c230:
Successfully signed ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example:trust-centos
```

Just like before, you'll be asked for Abe's signing password at the end of the push, and you should be able to see the 'Signed' badge beside the `:trust-centos` tag in the DTR web frontend.

9. Back on ucp-node-0, inspect the content trust metadata for the demo/example repo one more time:

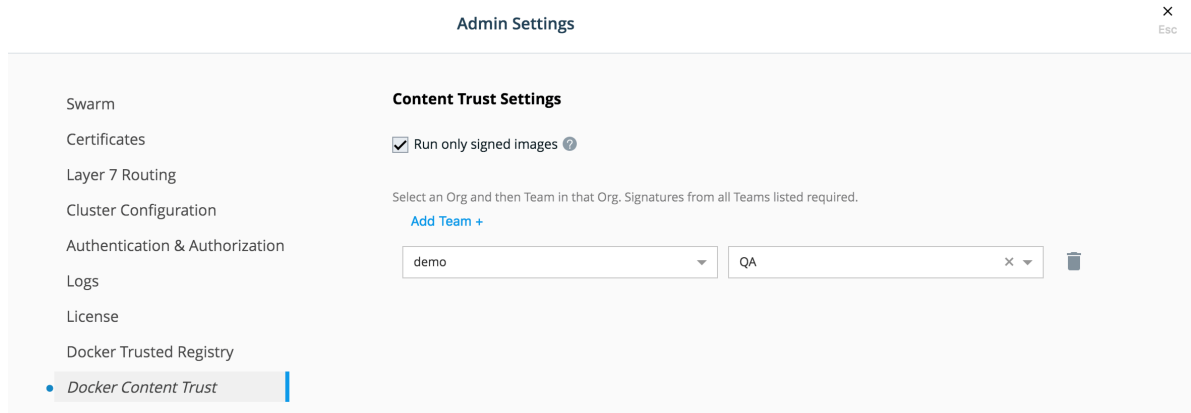
```
[centos@ucp-node-0 ~]$ docker trust inspect ${DTR_FQDN}:4443/demo/example
```

The output is similar to before, but now indicates that `:trust-centos` is a trusted tag that can be pulled with content trust enforced.

23.4 Requiring Trust at Runtime

UCP can be set to enforce valid content trust for all containers at runtime, without which deployment will fail, as follows:

1. In the UCP web frontend as an administrator, navigate **admin** -> **Admin Settings** -> **Docker Content Trust**, and click the **Run only signed images** checkbox.
2. Next click **Add Team** +, and require a signature from someone in the QA team of the demo organization:



Don't forget to hit **Save** at the bottom when done. We'll just require the one signature in this example, but note you can keep adding more signature requirements, which will be AND'd together, allowing you to demand a signature from every person who needs to sign off on an image before it goes into production.

3. Create a Swarm service in UCP using the `${DTR_FQDN}:4443/demo/example:trust-centos` image; set the command to `ping 8.8.8.8`. This should work exactly as before.
4. Create another Swarm service in UCP using the `${DTR_FQDN}:4443/demo/example:untrusted` image; this will fail with an error message **'image did not meet required signing policy'**
5. Try doing the previous step again, with the `:trusted` tag. This still fails in the same way - even though the `:trusted` tag has signed metadata from the admin user, this still isn't good enough. The trust policy you created above requires a member of the QA team to sign off.
6. Delete the service you created.

23.5 Revoking Trust Metadata

In the event that a tag should no longer have signed metadata associated with it, trust metadata can be revoked as follows.

1. Revoke the trust metadata for `demo/example:trust-centos`:

```
[centos@ucp-node-0 ~]$ docker trust revoke ${DTR_FQDN}:4443/demo/example:trust-centos
```

Enter passphrase for admin key with ID 00729ba:

Successfully deleted signature for `ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example:trust-centos`

Notice this procedure requires authentication from someone with signing authority over this repository.

2. Use `docker trust inspect` as above to confirm that `:trust-centos` is no longer a signed tag in the `demo/example` repository.

23.6 Revoking Signing Authority

In the event that a user should no longer have signing authority over a repository, their privileges can be revoked as follows.

1. Revoke Abe's signing authority for the `${DTR_FQDN}:4443/demo/example` repository:

```
[centos@ucp-node-0 ~]$ docker trust signer remove abe ${DTR_FQDN}:4443/demo/example
```

Removing signer "abe" from `ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example`...

Enter passphrase for repository key with ID dad1fba:

Successfully removed abe from `ec2-35-173-184-38.compute-1.amazonaws.com:4443/demo/example`

Note in this case, the repository key credentials are required.

2. Do `docker trust inspect` for this repository again. Abe is no longer on the 'Signers' list.

23.7 Optional: Enforcing Trust Pinning

An individual Docker engine can be put into *trust pinning* mode, which refuses to pull or run images except those with trust metadata signed by a member of a whitelist of root keys.

1. Remove `ucp-node-0` from your swarm, and clean up all containers and volumes running there (we can treat this like any remote node trying to interact with our DTR):

```
[centos@ucp-node-0 ~]$ docker swarm leave
[centos@ucp-node-0 ~]$ docker container rm -f $(docker container ls -aq)
[centos@ucp-node-0 ~]$ docker volume rm $(docker volume ls -q)
[centos@ucp-node-0 ~]$ unset DOCKER_CONTENT_TRUST
[centos@ucp-manager-0 ~]$ docker node rm ucp-node-0
```

2. Find the name of your root signing key. You noted down its prefix above when you first did `docker trust sign` (my example was `2f602d2`). List the contents of the private key directory on the node you created this private key on (`ucp-node-0` as above):

```
[centos@ucp-node-0 ~]$ ls ~/.docker/trust/private/

0eabbe7213fbaf10dbb637221f84aad14365e5d41e011acef9e2e546e4613892.key
2f602d205a188b9a1ab3bae1ed5d1964ac494fea950ad8997280dcc08bbb4578.key
c23ae5df4dd7f4a6a580980090cdf377e81b06fcfdacba3bab7b7ff019b2e98.key
```

The `.key` file that matches the prefix you found earlier is the one you're looking for; the full string is the name of your root key (`2f602d205a188b9a1ab3bae1ed5d1964ac494fea950ad8997280dcc08bbb4578` in my example). If you're unsure, open the file; it'll say `role: root` near the top if you've picked the right one.

3. Edit your `/etc/docker/daemon.json` file to look like this, where `<root key>` is the string you just found (without the `.key` file extension at the end):

```
{
  "storage-driver": "overlay2",
  "content-trust": {
    "trust-pinning": {
      "root-keys": {
        "<DTR Public DNS>:4443/demo/example": ["<root key>"]
      }
    },
    "mode": "enforced"
  }
}
```

4. Restart the Docker Engine to load this new config:

```
[centos@ucp-node-0 ~]$ sudo service docker restart
```

5. Now try and pull your `:trusted` tag from your DTR:

```
[centos@ucp-node-0 ~]$ docker image pull ${DTR_FQDN}:4443/demo/example:trusted

sha256:xxx: Pulling from demo/example
Digest: sha256:xxx
Status: Image is up to date for <DTR>:4443/demo/example:trusted@sha256:xxx
```

The pull works, since this image was signed using the root key specified.

6. Try pulling the `:untrusted` tag; the pull fails, much like it did when you were demanding any signature be available with `DOCKER_CONTENT_TRUST=1`.

- Now try pulling `alpine:latest`:

```
[centos@ucp-node-1 ~]$ docker image pull alpine:latest
Error response from daemon: could not validate the path to a trusted root: invalid trust pinning sp
```

This time, even pulling this official image will fail. While you were able to pull `alpine` in generic trust enforcing mode (`DOCKER_CONTENT_TRUST=1`), trust pinning only accepts signatures from a *whitelist* of signers. Try setting mode to disabled in `/etc/docker/daemon.json`, restart the Docker engine, set `DOCKER_CONTENT_TRUST=1`, and pull `alpine` again to illustrate the difference between the two trust enforcement modes.

- Clean up `ucp-node-0`:

- make sure mode is set to disabled in `/etc/docker/daemon.json`, then:
- `sudo service docker restart`
- `unset DOCKER_CONTENT_TRUST`
- Rejoin `ucp-node-0` to the UCP cluster.

23.8 Conclusion

In this exercise, we stepped through the process of establishing, enforcing, granting and revoking content trust. Content trust enforcement was originally designed around the TUF framework for protecting image downloads from man-in-the-middle attacks, but often finds primary application in even the most secure datacenters as a means of enforcement on what images can be run in UCP. By requiring sign-off from all stakeholders, content trust allows us to automate the enforcement of these governance processes.

24 Image Promotion & Webhooks

By the end of this exercise, you should be able to:

- Copy images from one repository or registry to another with image promotion and mirroring, respectively
- Fire a webhook at a custom URL in automatic response to an event in DTR
- Audit repository and registry level event logs to reconstruct events taken by DTR, including but not limited to the above.

24.1 Creating a Promotion Pipeline

- Create three repositories: `admin/whale-dev`, `admin/whale-qa`, and `admin/whale-vulnerable`.
- Set the `admin/whale-dev` repository to scan on push.
- Navigate **Repositories** -> `admin/whale-dev` -> **Promotions** -> **New Promotion Policy**, and create a policy that promotes images to `admin/whale-qa` if they have zero critical vulnerabilities. Also make sure the promoted image changes the tag to `qa-*`, where the asterisk is whatever the original tag of the image was.
- In the same repository, create another policy that promotes images that *do* have a critical vulnerability to `admin/whale-vulnerable`. Retag these images with the original tag, plus the current date.
- Visit `admin/whale-qa` and `admin/whale-vulnerable`'s Promotions tab, and click on *Is Target*; you should see the policies you just created that push images to each repository.

24.2 Setting up a Webhook

- On `ucp-manager-0`, create a service that will catch and display the webhook's POST payload (in reality, this would be some tooling like a build manager or notification system that needs to take action when the webhook fires, but for the sake of simplicity we'll just create a service that logs the POST information):

```
[centos@ucp-manager-0 ~]$ docker service create --name whale-webhook \
  -p 8000:8000 training/whale-server
```

2. Back in DTR, navigate **Repositories** -> **admin/whale-dev** -> **Webhooks** -> **New Webhook**.
3. Choose 'Tag Pushed to Repository' for the *notification to receive*, and provide `http://${ucp-manager-0_FQDN}:8000` as the webhook URL. Now whenever an image is pushed to this repository, DTR will POST some JSON describing the newly pushed image to the URL you provided.
4. Save this webhook by clicking **Create**.

24.3 Triggering the Pipeline

1. Find an image on Docker Hub that has some critical vulnerabilities, eg `ubuntu:16.04`. Pull it down, tag it as `${DTR_FQDN}:4443/admin/whale-dev:0.1`, and push it to DTR.
2. Check the logs of the `whale-webhook` service:

```
[centos@ucp-manager-0 ~]$ docker service logs whale-webhook
```

You should see the POST request generated by the webhook when it was triggered by pushing to the `whale-dev` repository.

3. Wait for the image scan on your `whale-dev:0.1` to complete, and then make sure the promotion to `admin/whale-vulnerable` worked as expected.
4. Find an image on Docker Hub that has *no* critical vulnerabilities, eg `busybox:latest` or `alpine:latest`, tag it as `${DTR_FQDN}:4443/admin/whale-dev:0.2`, and push it to DTR. Check that it was promoted as expected after its scan finishes, and look at the logs of `whale-webhook` once more to see the result of the webhook firing from this push.

24.4 Auditing DTR Events

Now that we've set DTR up to run a number of automated procedures, we'd like to be able to audit logs of actions taken.

1. In DTR, navigate to **Repositories** -> **admin/whale-dev** -> **Activity**. Here we see a list of all the events taken in this repository, both initiated by users and taken automatically by DTR. Use it to answer the following questions:
 - At what time was a 'Scan Tag' event initiated for `admin/whale-dev:0.1`?
 - At what time was `whale-dev:0.1` last promoted?
 - Who was the last user to initiate an 'Update Tag' event on the `admin/whale-dev:0.1` tag?
2. Similarly, system-level events can be audited from the job logs. In DTR, navigate to **System** -> **Job Logs**, and use the logs there to answer the following questions:
 - At what time was the last webhook fired? What endpoint was it sent to? (Hint: notice the 'View Logs' link on the right of the Job Logs table).
 - How many webhooks has DTR fired?
 - When was the last time an 'update_vuln_db' job was run?
 - How often are 'tag_prune' jobs run?

24.5 Optional: Image Mirroring

DTR can promote an image not only to another repository it controls, but to a repository in an entirely separate deployment of DTR; we refer to this as 'image mirroring'. If feasible, try this with a partner; use one partner's DTR as 'DTR 1', and the other's deployment as 'DTR 2'.

1. Login as admin to DTR 1 and to DTR 2.

2. On DTR 1 create a public repository `admin/mirror-origin`.
3. On DTR 2 create a public repository `admin/mirror-target`.
4. Back on DTR 1, navigate **Repositories** -> `admin/mirror-origin` -> **Mirrors** -> **New mirror**, fill in the form with connection info for DTR 2, and specify `admin/mirror-target` as the repository to mirror to.
5. Under **Show advanced settings** select **SKIP TLS**.
6. Do not add any triggers at this time. This will cause all images pushed to the repo on DTR 1 to be mirrored to the target on DTR 2.
7. Click **Connect** and then **Save and Apply** on the bottom of the view.
8. In the CLI e.g. tag an alpine image to `<DTR_1_FQDN>:4443/admin/mirror-origin:1.0`.
9. Push the above image.
10. Demonstrate that the image is available in DTR 1, and has been mirrored to DTR 2.

24.6 Conclusion

In this exercise, you set up a basic DTR pipeline. Images can be promoted or mirrored from one repository to another based on conditions that reflect progress through your CI/CD, review, and approval process, allowing you to create a clear picture of how an image moves from development to production, and understand precisely where a given image is in that process. Furthermore, webhooks can be fired at any point in this process to integrate with the rest of your tooling.

25 Tag Pruning and Garbage Collection

By the end of this exercise, you should be able to:

- Configure tag pruning and garbage collection in DTR
- Establish a pruning policy that reflects company auditing requirements

25.1 Pruning Tags

1. In a fresh directory `~/prune` on `ucp-manager-0`, make a Dockerfile with the following content:

```
FROM centos:7

RUN yum update -y
RUN yum install -y nano
```

Make sure `$DTR_FQDN` is set to the appropriate public DNS (either `infra` if you set up a DTR load balancer there, or `ucp-manager-0` if not), and build your image as usual with `docker image build -t ${DTR_FQDN}:4443/admin/prune:0.1 ..`. Push this up to DTR.

2. Make another version of your image by changing `nano` to `vim` in the above Dockerfile, tag it as `${DTR_FQDN}:4443/admin/prune:0.2`, and push to DTR.
3. In DTR, navigate **Repositories** -> `admin/prune` -> **Tags**; both your tags should be visible.
4. Navigate to the **Settings** tab in the same view, and scroll down to the *Pruning* section. Set the tag limit field to 1, and click **Save**.
5. Return to the **Tags** tab. Your 0.1 tag has been deleted by your pruning policy, which only keeps the latest tag pushed to this repository. *Note the time* - we'll need to know roughly when this tag was deleted for the garbage collection step later in this exercise.
6. Navigate **System** -> **Job Logs** and look for a `tag_prune` job near the top of the list; logs from the prune operation are available here.

25.2 Configuring Pruning Policies

In the last section, we set a simple limit on the number of tags allowed in a repository; we can set more sophisticated pruning policies as follows.

1. Turn off the limit on number of tags by setting the tag limit field you modified above back to 0.
2. Navigate **Repositories** -> **admin/prune** -> **Pruning**, and click **New pruning policy**.
3. Your company requires images to be retained for 12 months for auditing purposes, after which they are safe to delete. Configure a pruning policy that does so here, and then click **Prune all tags** to apply the policy to all current and future tags.

25.3 Configuring Garbage Collection

1. Determine how much space images are currently consuming on disk. If you're using minio, check the size of its backing volumes on your infra node:

```
[centos@infra ~]$ sudo du -h --max-depth 1 /var/lib/docker/volumes | grep minio

118M    /var/lib/docker/volumes/minio_stack_minio2-data
118M    /var/lib/docker/volumes/minio_stack_minio3-data
118M    /var/lib/docker/volumes/minio_stack_minio1-data
118M    /var/lib/docker/volumes/minio_stack_minio4-data
```

2. Make sure it's been at least 5 minutes since you deleted the 0.1 tag of the admin/prune repository. Garbage collection only cleans up layers that have been untagged for at least five minutes.
3. In DTR, navigate **System** -> **Garbage Collection**, and select:
 - **Until Done**, allowing GC to proceed until complete
 - **Daily at Midnight UTC** from the dropdown, so GC is run every night
 - Click **Save and Start** to establish your GC policy and initiate a garbage collection job immediately.
4. Navigate to **System** -> **Job Logs** and look for an *onlinegc* job near the top of the list; once this job reports done, garbage collection is complete.
5. Check the size on disk of your DTR storage backing again, the way you did above; some disk space should have been freed up by garbage collection. Alternatively, try re-pushing admin:prune:0.1; you should see at least one layer needs to be re-uploaded, since it was deleted by garbage collection.

Remember, garbage collection only deletes layers that aren't referenced by *any* tag. If the above steps didn't free up any space, you might have some other tag pointing to the same image layers. Try deleting a few repositories, wait five minutes, and garbage collect again.

25.4 Conclusion

In this exercise, we saw how to configure automatic tag pruning and garbage collection in order to automatically mitigate the amount of tags and disk space occupied by DTR-managed images. Defining a useful tag pruning policy usually rests on understanding your company's software auditing and retirement policies, specifically with respect to image age; establish when an image is either old enough or enough versions out of date to be retired, and build pruning policy around those decision. In terms of garbage collection, bear in mind this can be a resource-intensive process; consider using the interface we toured above to limit how much time garbage collection is allowed to run for, and to schedule it for off-peak hours for your DTR.

26 (Optional) Content Caching

By the end of this exercise, you should be able to:

- Establish a DTR content cache
- Help a DTR user configure their user account and local Docker Engine to pull from a cache of their choice

26.1 Setup

DTR content caches are meant to run on independent swarms, geographically close to DTR users who intend to use it. For this exercise, we'll make an independent cache swarm by removing one of our UCP worker nodes and setting it up as its own swarm.

1. Remove ucp-node-0 from your swarm, clean it up, and establish it as its own swarm:

```
[centos@ucp-node-0 ~]$ docker swarm leave
[centos@ucp-node-0 ~]$ docker container rm -f $(docker container ls -aq)
[centos@ucp-node-0 ~]$ docker swarm init
```

2. Make sure the removal is clean by deleting the node from the UCP manager consensus via ucp-manager-0:

```
[centos@ucp-manager-0 ~]$ docker node rm ucp-node-0
```

26.2 Establishing a Content Cache

1. Label ucp-node-0 as your content cache host (not strictly necessary in a single node swarm, but if this were running on a real local cluster, we'd like to pin the cache to a specific machine users can connect to):

```
[centos@ucp-node-0 ~]$ docker node update --label-add dtr.cache=true ucp-node-0
```

2. On ucp-node-0, create a file called ~/docker-stack.yml with the following content (also downloadable from <https://bit.ly/2CKP7Of>):

```
version: "3.3"
services:
  cache:
    image: docker/dtr-content-cache:2.6.0-beta3
    entrypoint:
      - /start.sh
      - "/config.yml"
    ports:
      - 443:443
    deploy:
      replicas: 1
      placement:
        constraints: [node.labels.dtr.cache == true]
      restart_policy:
        condition: on-failure
    configs:
      - config.yml
    secrets:
      - dtr.cert.pem
      - cache.cert.pem
      - cache.key.pem
  configs:
    config.yml:
      file: ./config.yml
  secrets:
    dtr.cert.pem:
      file: ./certs/dtr.cert.pem
    cache.cert.pem:
      file: ./certs/cache.cert.pem
```

```
cache.key.pem:
  file: ./certs/cache.key.pem
```

3. Also on ucp-node-0, create a file called `~/config.yml` with the following content, downloadable from <https://bit.ly/2CgWuw0>. Note the `<variables>` that need to be replaced with values for your cluster:

```
version: 0.1
log:
  level: info
storage:
  delete:
    enabled: true
  filesystem:
    rootdirectory: /var/lib/registry
http:
  addr: 0.0.0.0:443
  secret: generate-random-secret
  host: https://<ucp-node-0 FQDN>
  tls:
    certificate: /run/secrets/cache.cert.pem
    key: /run/secrets/cache.key.pem
middleware:
  registry:
    - name: downstream
    options:
      blobttl: 24h
      upstreams:
        - https://<DTR FQDN>:4443
      cas:
        - /run/secrets/dtr.cert.pem
```

4. Make a directory `~/certs`. From within that directory, fetch your DTR certificate:

```
[centos@ucp-node-0 ~]$ mkdir certs ; cd certs
[centos@ucp-node-0 certs]$ DTR_FQDN=<DTR FQDN>
[centos@ucp-node-0 certs]$ sudo curl -k https://${DTR_FQDN}:4443/ca -o dtr.cert.pem
```

5. Still within `~/certs`, generate a keypair with a self-signed cert. You will be asked a bunch of questions; answer . to all of them except the Common Name (eg, your name or your server's hostname); for this, provide the FQDN of ucp-node-0:

```
[centos@ucp-node-0 certs]$ openssl req -newkey rsa:2048 -nodes \
  -keyout key.pem -x509 -days 365 -out certificate.pem
```

```
Generating a 2048 bit RSA private key
```

```
.....+++
.....+++
```

```
writing new private key to 'key.pem'
```

```
-----
```

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
```

```
What you are about to enter is what is called a Distinguished Name or a DN.
```

```
There are quite a few fields but you can leave some blank
```

```
For some fields there will be a default value,
```

```
If you enter '.', the field will be left blank.
```

```
-----
```

```
Country Name (2 letter code) [XX]:.
```

```
State or Province Name (full name) []:.
```

```

Locality Name (eg, city) [Default City]:.
Organization Name (eg, company) [Default Company Ltd]:.
Organizational Unit Name (eg, section) []:.
Common Name (eg, your name or your server's hostname) []:<ucp-node-0 FQDN>
Email Address []:.

```

If successful, you should now have two files, `certificate.pem` and `key.pem`. Move them into position:

```

[centos@ucp-node-0 certs]$ mv certificate.pem cache.cert.pem
[centos@ucp-node-0 certs]$ mv key.pem cache.key.pem

```

6. Change directory back to where you put your stack file `docker-stack.yml`, deploy the cache on `ucp-node-0`, and confirm it is running:

```

[centos@ucp-node-0 certs]$ cd ~
[centos@ucp-node-0 ~]$ docker stack deploy --compose-file docker-stack.yml dtr-cache
[centos@ucp-node-0 ~]$ docker stack ps dtr-cache

```

```

... NAME                                IMAGE                                NODE            ... CURRENT STATE
... dtr-cache_cache.1                  docker/dtr-content-cache:2.6.0-beta3 ucp-node-0      ... Running 36 minutes a

```

7. From an admin account in DTR, navigate **API -> POST -> Try It Out** under the **content_caches** heading. Fill in the **body** with the following JSON:

```

{
  "host": "https://<ucp-node-0 FQDN>:443",
  "name": "mycache"
}

```

Click **Execute** to register your new cache in DTR.

26.3 Pulling from a Content Cache

1. From an admin account in DTR, navigate **Users -> <username> -> Settings** (any user will do), and scroll down to the 'Content Cache' section. Choose your new content cache 'mycache' out of the 'Region' dropdown, and click **Save**.
2. On `ucp-node-1`, establish certificate trust with your cache by copying `~/certs/cache.cert.pem` (from `ucp-node-0`) to `/etc/pki/ca-trust/source/anchors/<ucp-node-0 FQDN>:443.crt`, and trust the certificate:

```

[centos@ucp-node-1 ~]$ sudo update-ca-trust
[centos@ucp-node-1 ~]$ sudo /bin/systemctl restart docker.service

```

3. Still on `ucp-node-1`, log in to DTR as before, as the user you established the cache for (you'll need to establish certificate trust with the upstream DTR as well as the cache to do this; see the 'Installing DTR' exercise for instructions). Pull any image from your DTR.
4. On your cache `ucp-node-0`, confirm that the image you just pulled was routed through the cache by checking the logs of the cache service:

```

[centos@ucp-node-0 ~]$ docker service logs dtr-cache_cache

```

You should be able to find logs referencing the image you just pulled.

5. We can further confirm that the image we pulled passed through our cache by finding it in the cache container's filesystem:

```

[centos@ucp-node-0 ~]$ docker container exec -it <cache container ID> sh
/# ls /var/lib/registry/docker/registry/v2/repositories

```

You should see the owner (user or organization) of the image you pulled; list that directory, and you'll see the name of the image, and below that, the layers themselves.

26.4 Conclusion

In this exercise, you established a content cache, configured a user account to route pulls through it, and set up a Docker engine to trust your cache and pull from it. In addition to mitigating bandwidth consumption for remote users, a content cache can help limit requests to your upstream DTR by serving images from satellite locations.

27 The Software Supply Chain

When combined with a build agent and source control service, Universal Control Plane and Docker Trusted Registry form a powerful pipeline for containerized software. In this challenging exercise, you will use what you learned in this workshop to try and build a minimal working example of such an automated pipeline.

Please organize yourself in teams of 2.

27.1 Your Pipeline

We want to build a software pipeline that takes the following steps:

- Pipeline is triggered when code is pushed to version control.
- Build agent will pull code from version control and a base image from DTR, build the Dockerfile found in version control, and test the image using unit tests provided.
- When tests pass, build agent will push image to a Docker Trusted Registry being used as a development and QA sandbox.
- Upon receipt of an image, the development DTR will automatically scan the image; if no vulnerabilities are found, the image is promoted to a QA repository in the same DTR.
- A webhook will trigger the build agent to deploy the app from the QA repository upon receipt, into a QA environment.
- If the app successfully runs in the QA environment, a human will use content trust to sign the images and push them to a production-only DTR.
- Upon receipt in the production DTR, a final webhook is fired telling the build manager to deploy the app in the production environment. UCP should enforce the presence of a signature from the QA manager in this environment.

The steps below will walk through setting this pipeline up in a little more detail, but not much - it's up to you to assemble what you learned in this workshop to successfully build a pipeline like the one above.

Note that the instructions below prescribe particular tooling (GitHub; Jenkins) and steps for building your pipeline. This is not mandatory; use whatever tooling you prefer to build your software pipeline and be creative, so long as your solution performs the tests, sign-offs, promotions and environment separations described above.

27.2 Tasks

1. Person 1 contributes their lab as Dev environment. Person 2 contributes their lab as Prod environment.
2. Fork repository `/docker-training/dops-final-project` on GitHub.
3. Analyze the code of this simple web project. It is a photo album showing some animals and will be used as the application that we will build, test and deploy.
4. Deploy a containerized version of Jenkins to your `infra` node in your Dev environment.

Note: If you're not familiar with Jenkins you can get inspired by the optional exercise "Appendix: Build Server" in your exercise book. This exercise walks through a sample setup of Jenkins step-by-step. Alternatively, feel free to use or create an alternate build management tool to achieve the same results.

5. Configure Jenkins and your source control repository so that when new code is pushed, Jenkins does the following:
 - pulls the latest code
 - builds that code into an image using the Dockerfile it finds at the root of that repository
 - runs unit tests also found in that repository against this new image
 - if the build is successful and the unit tests pass, then have Jenkins push the image to the Dev DTR.
6. Set the repository in Dev that Jenkins pushes to to scan automatically. If the image scan detects no vulnerabilities, automatically promote the image to the QA repository in Dev DTR.
7. The QA repo should have a webhook that triggers Jenkins to deploy the app into the Dev environment. Visit the app in this environment to make sure all is running well.
8. If you're satisfied with your app in the Dev environment, one person should manually sign the image and push it to the Prod DTR. Make sure to sign with a delegation key, and with server-side keys managed in the Prod DTR.
9. Configure the Prod UCP to only run images signed by the person who signed your image in the last step.
10. Define a webhook on the prod repository to trigger Jenkins to deploy the application to the Prod environment.
11. Update something in the application code to trigger the pipeline.

27.3 Optional Challenges

1. Migrate the application to Kubernetes.
2. Define health checks (liveness and readiness probes) for the application.
3. Update the application code and deployment configuration again, this time with an error to trigger a rollback during deployment.

27.4 Conclusion

In this exercise, you set up a proof-of-concept software supply chain. There are of course many ways to build such a pipeline, but they should all share the characteristics of image integrity and provenance, build and test automation, and environment isolation aspired to above, in order to provide secure container development and deployment at scale.

28 Appendix: Build Server

In a typical Docker pipeline, a build agent will be responsible for pulling code and base images from their respective registries, building developers' images, and pushing them to DTR to trigger a CI/CD pipeline. By the end of this exercise, you should be able to:

- configure a containerized Jenkins deployment to watch a GitHub repo and build images out of code pushed there
- automatically push Jenkins-built images to DTR, potentially kicking off scanning and promotions from there.

A lot of information has been distilled from here: <https://github.com/yongshin/leroy-jenkins>.

28.1 Prerequisites

1. Make sure all your UCP swarm nodes trust DTR, if you haven't already:

```
[centos@ucp-x ~]$ DTR_FQDN=<Public DNS DTR is reachable at>
[centos@ucp-x ~]$ sudo curl -k https://$DTR_FQDN:4443/ca \
-o /etc/pki/ca-trust/source/anchors/$DTR_FQDN:4443.crt
[centos@ucp-x ~]$ sudo update-ca-trust
[centos@ucp-x ~]$ sudo /bin/systemctl restart docker.service
```

28.2 Creating and Shipping the Jenkins Image

1. On ucp-node-1, create a folder build-server/jenkins.
2. Add a Dockerfile to the folder with this content:

```
FROM jenkins:2.60.3
USER root
RUN apt-get update \
  && apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common \
  && curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add - \
  && add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/debian \
    $(lsb_release -cs) \
    stable" \
  && apt-get update \
  && apt-get install -y docker-ce sudo \
  && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers
COPY entrypoint.sh /
ENTRYPOINT /entrypoint.sh
```

we're using the base Jenkins image, and adding Docker CE and a few other libraries to it.

3. Add a file entrypoint.sh to the same folder with the following content:

```
openssl s_client -connect ${DTR_IP}:443 -showcerts \
  </dev/null 2>/dev/null | openssl x509 -outform PEM | sudo tee \
  /usr/local/share/ca-certificates/${DTR_IP}.crt
sudo update-ca-certificates

/bin/tini -- /usr/local/bin/jenkins.sh
```

4. Make the above file executable:

```
[centos@ucp-node-1 jenkins]$ chmod +x ./entrypoint.sh
```

5. Build the image, and push to DTR:

```
[centos@ucp-node-1 jenkins]$ DTR_FQDN=<Public DNS DTR is reachable at>
[centos@ucp-node-1 jenkins]$ docker image build \
  -t ${DTR_FQDN}:4443/admin/my-jenkins:1.0 .
[centos@ucp-node-1 jenkins]$ docker image push ${DTR_FQDN}:4443/admin/my-jenkins:1.0
```

28.3 Preparing the Repository in DTR

1. Login to your DTR as admin.
2. Create a public repo called admin/jenkins-demo. This is where Jenkins will push built images to.

28.4 Preparing a Source Repo

1. Log in to GitHub, and create a public repo called jenkins-demo. This is where the code describing your image will be controlled.
2. On your local machine or infra node, make a fresh folder called ~/jenkins-demo, and add a Dockerfile to it describing an image you're developing (feel free to just use the Dockerfile and entrypoint.sh defined above for the Jenkins image itself if you like).
3. Set this code up to push, but *don't* push yet:

```
[centos@infra ~]$ cd jenkins-demo
[centos@infra jenkins-demo]$ git init
[centos@infra jenkins-demo]$ git remote add origin <your jenkins-demo GitHub repo url>
[centos@infra jenkins-demo]$ git add *
[centos@infra jenkins-demo]$ git commit -m 'code ready for pipeline testing'
```

Note: When you commit, you might be asked by git to identify yourself. Feel free to skip this step for this exercise, or provide your GitHub username and corresponding email if you like (not required).

28.5 Running Jenkins in the Swarm

28.5.1 Preparing the Swarm Node

1. SSH into the swarm node ucp-node-1 where we'll run Jenkins.
2. Create a directory ~/jenkins and a directory ~/admincerts:

```
[centos@ucp-node-1 ~]$ mkdir jenkins
[centos@ucp-node-1 ~]$ mkdir admincerts
```

3. Create an auth token to use UCP's API from ucp-node-1:

```
[centos@ucp-node-1 ~]$ UCP_FQDN=<Public DNS UCP is reachable at>
[centos@ucp-node-1 ~]$ AUTHTOKEN=$(curl -sk \
  -d '{"username":"admin","password":"adminadmin"}' \
  https://${UCP_FQDN}/auth/login | jq -r .auth_token)
```

4. Download and unzip a client bundle for your admin account:

```
[centos@ucp-node-1 ~]$ cd admincerts
[centos@ucp-node-1 admincerts]$ curl -k -H "Authorization: Bearer $AUTHTOKEN" \
  https://${UCP_FQDN}/api/clientbundle -o bundle.zip
[centos@ucp-node-1 admincerts]$ unzip bundle.zip
```

5. Run the shell configuration script which is part of the bundle:

```
[centos@ucp-node-1 admincerts]$ eval "$(<env.sh)"
```

28.5.2 Creating the Jenkins Service

1. SSH into ucp-manager-0.
2. Label ucp-node-1 for Jenkins so we can use scheduling constraints to place the Jenkins master on that node:

```
[centos@ucp-manager-0 ~]$ docker node update --label-add jenkins=master ucp-node-1
```

3. Run a service for Jenkins:

```
[centos@ucp-manager-0 ~]$ export DTR_FQDN=<public DNS DTR is reachable at>
[centos@ucp-manager-0 ~]$ docker service create \
  --name my-jenkins --publish 8080:8080 \
  --mount type=bind,source=/home/centos/jenkins,destination=/var/jenkins_home \
  --mount \
  type=bind,source=/home/centos/admincerts,destination=/home/jenkins/admincerts \
  --constraint 'node.labels.jenkins == master' \
  --detach=false \
  -e DTR_IP=${DTR_FQDN} \
  ${DTR_FQDN}:4443/admin/my-jenkins:1.0
```

4. Verify that the service is up and running using e.g.:

```
[centos@ucp-manager-0 ~]$ docker service ps my-jenkins
```

28.5.3 Finalizing the Jenkins Configuration

1. Open a browser window and navigate to Jenkins at `http://<any UCP public IP>:8080`
2. Enter the initial password that you can get on ucp-node-1 by entering this command:

```
[centos@ucp-node-1 ~]$ sudo cat ~/jenkins/secrets/initialAdminPassword
```

3. Install the suggested plugins from the initial popup window, create your admin account and click **Continue as Admin -> Start Using Jenkins**.

28.6 Configuring Jenkins Jobs

1. Login to Jenkins if not already done.
2. Create a Jenkins Job of type **Free Style** and call it **docker build and push**.
3. In **Source Code Management** -> **Git** - set repository to the url of the jenkins-demo GitHub repo you created above.
4. Set Build Triggers -> Poll SCM, and enter * * * * * in the 'Schedule' field, to poll for code changes every minute. See <https://en.wikipedia.org/wiki/Cron> for an explanation of this scheduling syntax.
5. Add a Build Step of type **Execute Shell** and add the following script (remember to replace <DTR FQDN> with the public DNS you're reaching DTR at):

```
#!/bin/bash
cd /home/jenkins/admincerts
eval "$(<env.sh)"
cd -
export DTR_FQDN=<DTR FQDN>
docker image build --build-arg constraint:node==ucp-node-1 \
  -t ${DTR_FQDN}:4443/admin/jenkins-demo:1.${BUILD_NUMBER} .
docker login -u admin -p adminadmin ${DTR_FQDN}:4443
docker image push ${DTR_FQDN}:4443/admin/jenkins-demo:1.${BUILD_NUMBER}
```

Notice the combination of the client bundle with the `--build-arg constraint:node==ucp-node-1` flag to `docker image build`; this allows our containerized Jenkins to build an image on the local machine, without mounting the docker socket.

6. Save the job.

7. Back on your development machine, push your jenkins-demo code to GitHub (`git push origin master` in the jenkins-demo folder). Wait a minute, and SCM polling in Jenkins will trigger a build and push your new image to DTR when complete.

28.7 Conclusion

In this exercise we set up a containerized Jenkins build server that ingests code from a version control repository, and pushes built images into Docker Trusted Registry. From this point, you can configure DTR to scan images for vulnerabilities, promote them to new repositories, and trigger webhooks to external services at each step.

Instructor Demos

1 Instructor Demo: Containerized Nature of UCP

In this demo, we'll illustrate:

- The containerized nature of UCP manager and worker nodes
- The containerized nature of Kubernetes

1.1 Containers on UCP Manager and Worker Nodes

1. Now that you have UCP installed with one manager and 2 worker nodes, list the containers on the ucp-manager-0 node:

```
[centos@ucp-manager-0 ~]$ docker container ls
```

Your result should resemble this:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
e1da75aee4c2	docker/ucp-agent:3.1.1	"/bin/ucp-agent agent"	About an hour ago	Up A
2f429032f6f3	docker/ucp-hyperkube:3.1.1	"/bin/apiserver_entr..."	2 hours ago	Up 2
efde199c3e9c	docker/ucp-controller:3.1.1	"/bin/controller ser..."	2 hours ago	Up 2
bb6302db2657	docker/ucp-auth:3.1.1	"/usr/local/bin/enzi..."	2 hours ago	Up 2
71c7084c1985	docker/ucp-auth:3.1.1	"/usr/local/bin/enzi..."	2 hours ago	Up 2
...				

2. Furthermore, you can view the same list on your UCP by clicking **Shared Resources** -> **Nodes** -> **ucp-manager-0** -> **Metrics** then scroll down to see the list of all the containers making up UCP on this node:

Nodes

ucp-manager-0
● ready Created 2 hours ago, Last Updated: 2 hours ago

Overview Metrics Volumes Networks

STATE	STATUS	ID	NAME	IMAGE	CREATED
●	Up About...	e1da75c	/ucp-manager-0/ucp-agent.ojzhdft5sk3th7b5v504siam3.p4zojgftvsgdj	dockereng/	2 hours ago
●	Up About...	2f42903	/ucp-manager-0/ucp-kube-apiserver	dockereng/	2 hours ago
●	Up About...	efde199	/ucp-manager-0/ucp-controller	dockereng/	2 hours ago
●	Up About...	bb6302c	/ucp-manager-0/ucp-auth-api.ojzhdft5sk3th7b5v504siam3.nu45kicnk	dockereng/	2 hours ago
●	Up About...	71c7084	/ucp-manager-0/ucp-auth-worker.ojzhdft5sk3th7b5v504siam3.127k6	dockereng/	2 hours ago
●	Up About...	634c9d8	/ucp-manager-0/ucp-kube-scheduler	dockereng/	2 hours ago
●	Up About...	5584e3f	/ucp-manager-0/ucp-kube-controller-manager	dockereng/	2 hours ago
●	Up About...	704a2b2	/ucp-manager-0/ucp-swarm-manager	dockereng/	2 hours ago
●	Up About...	4a403d3	/ucp-manager-0/ucp-auth-store	dockereng/	2 hours ago

Notice how nothing has been deployed on your UCP yet, but containers already exist on these nodes. This is what we mean by stating that UCP is just a containerized application. Everything needed to run UCP is in these containers.

3. On a worker node, list all containers:

```
[centos@ucp-node-1 ~]$ docker container ls
```

You should see:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
0edfcf453757	docker/ucp-agent:3.1.1	"/bin/ucp-agent agent"	2 hours ago	Up 2
a81fbd9846b8	docker/ucp-hyperkube:3.1.1	"kubelet --allow-pri..."	2 hours ago	Up 2
042ce9876aa1	docker/ucp-hyperkube:3.1.1	"kube-proxy --cluste..."	2 hours ago	Up 2
025df6fc745a	docker/ucp-agent:3.1.1	"/bin/ucp-agent prox..."	2 hours ago	Up 2
...				

4. Just like manager nodes, you can view the same containers on UCP by navigating **Shared Resources -> Nodes -> ucp-node-1 -> Metrics** then scrolling down to see the list of all the containers making up UCP on this node:

Nodes

ucp-node-1
● ready Created 2 hours ago, Last Updated: 2 hours ago

Overview Metrics Volumes Networks

0.5

11:00 PM 12:00 AM 01:00 AM 02:00 AM 03:00 AM 04:00 AM

Actions

STATE	STATUS	ID	NAME	IMAGE	CREATED
●	Up About...	0edfcf4f	/ucp-node-1/ucp-agent.hkmicwml39snvf8dn4i3pmx9d.tg51avb5362y	dockereng/	2 hours ago
●	Up About...	a81fbd9	/ucp-node-1/ucp-kubelet	dockereng/	2 hours ago
●	Up About...	042ce9e	/ucp-node-1/ucp-kube-proxy	dockereng/	2 hours ago
●	Up About...	025df6f	/ucp-node-1/ucp-proxy	dockereng/	2 hours ago

While there are less containers running on worker nodes, you can still see that UCP on these nodes, just like on manager nodes, is made up of containers.

1.2 Containerized Nature of Kubernetes

One of UCP's main responsibilities is to bootstrap Kubernetes; it does so by containerizing all of its components, and deploying and managing them alongside the rest of UCP's containers.

1. On your manager node, list all of the containers that make up Kubernetes on UCP:

```
[centos@ucp-manager-0 ~]$ docker container ls | grep "kube"
```

You should see something like this:

2f429032f6f3	docker/ucp-hyperkube:3.1.1	"/bin/apiserver_entr..."	3 hours ago	Up 3
634c9d8c9982	docker/ucp-hyperkube:3.1.1	"kube-scheduler --ku..."	3 hours ago	Up 3
5584e354da0d	docker/ucp-hyperkube:3.1.1	"kube-controller-man..."	3 hours ago	Up 3
4a10c058e535	docker/ucp-hyperkube:3.1.1	"kubelet --allow-pri..."	3 hours ago	Up 3
6711f2f860d9	docker/ucp-hyperkube:3.1.1	"kube-proxy --cluste..."	3 hours ago	Up 3
...				

Just like the previous section, you can navigate to **Shared Resources -> Nodes -> ucp-manager-0 -> Metrics** and scroll down to the list of containers, search for kube, and be able to see all of the containers needed for Kubernetes to run on UCP:

1.3 Supporting Resources

In addition to its containers, UCP uses a number of services, volumes, and configurations, all managed automatically by Docker.

1. List your services on one of your managers:

```
[centos@ucp-manager-0 ~]$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
ilkwicppwfg1	ucp-agent	global	3/3	docker/ucp-agent
1ul9kbh2r81v	ucp-agent-s390x	global	0/0	docker/ucp-agent
dury1nd9aaia	ucp-agent-win	global	0/0	docker/ucp-agent
dkkre1ya73mc	ucp-auth-api	global	1/1	docker/ucp-auth
p3tz9geqw3ta	ucp-auth-api-s390x	global	0/0	docker/ucp-auth

oa45vjrb0tjy	ucp-auth-worker	global	1/1	docker/ucp-auth
lzefx9dmqyab	ucp-auth-worker-s390x	global	0/0	docker/ucp-auth

A number of services exist:

- `ucp-agent-*` is a global service that manages most other UCP containers. We have three nodes in our cluster right now, so we have three replicas of this service.
- `ucp-auth-api-*` and `ucp-auth-worker-*` run only on managers (which we currently have only one of), and handle authentication and authorization requests.
- `*-win` services run on Windows workers, which we currently have zero of in this cluster
- `*-s390x` services run on IBM mainframe workers, which we also have none of at the moment.

2. List your volumes on a manager:

```
[centos@ucp-manager-0 ~]$ docker volume ls
```

DRIVER	VOLUME NAME
local	02674424e2b66d8386f6ed394e38240412cc270c21da967d2d589784f7a8b082
local	ucp-auth-api-certs
local	ucp-auth-store-certs
local	ucp-auth-store-data
local	ucp-auth-worker-certs
local	ucp-auth-worker-data
local	ucp-client-root-ca
local	ucp-cluster-root-ca
local	ucp-controller-client-certs
local	ucp-controller-server-certs
local	ucp-kv
local	ucp-kv-certs
local	ucp-metrics-data
local	ucp-node-certs

Most UCP volumes exist to hold certificates for mutual TLS authentication between UCP components, but a few such as `ucp-kv` and `ucp-auth-store-data` serve as the storage backend for UC's databases.

3. Finally, list your configurations on a manager:

```
[centos@ucp-manager-0 ~]$ docker config ls
```

ID	NAME	CREATED
emjnddwvarrhj2pwt7946ho1c	com.docker.license-0	20 minutes ago
16hs2p68gxrc6bh29haz8jf6l	com.docker.ucp.internal-config.LRQ59MG5L3LF9HCRMTA7F34A	27 minutes ago

In addition to the license and configuration file we see here, UCP will write other configs depending on usage, such as for managing load balancer configuration.

1.4 Conclusion

Docker's UCP is a containerized application built entirely out of Docker components running on a Swarm. When UCP is deployed, it starts running the globally scheduled service called `ucp-agent`, which monitors the node it runs on and maintains the rest of the UCP components co-located there. In addition to its containers, UCP also manages several volumes and configurations; therefore, when deleting UCP, it's important to wipe out all of these objects in the correct order: services, then containers, then volumes, then configurations.

2 Instructor Demo: UCP RBAC

In this demo, we'll illustrate:

- Constructing Swarm resource collections and Kubernetes namespaces through UCP
- Constructing users, teams and organizations in UCP
- Granting permissions to users through permission grants and role bindings

2.1 Part 1: Basic Swarm RBAC

1. Start by creating a user named Moby: navigate **Access Control** -> **Users** -> **Create**, and fill out the form. Note usernames must be all lowercase, and passwords must be at least 8 characters long.
2. Create a resource collection: navigate **Shared Resources** -> **Collections** -> **View Children**, and click **Create Collection**. Name the resource collection *bigstore*.
3. Create a grant giving Moby the *Full Control* role over the bigstore resource collection: navigate **Access Control** -> **Grants** -> **Swarm** -> **Create Grant**, and fill out the corresponding fields.
4. Log in to UCP as Moby, and create a service in the bigstore resource collection; it proceeds as you'd expect, since Moby has full control over this resource collection. Try and create another in the *Shared* or *Private* resource collections, and it will fail since Moby has no rights to these collections.

2.2 Part 2: Resource Collections, Teams, and Organizations

1. Delete Moby's grant from the last section: log back into UCP as admin, navigate **Access Control** -> **Grants** -> **Swarm**, check the checkbox beside Moby's grant to bigstore, and select **Actions** -> **Remove**.
2. Create two sub-collections in the bigstore collection: *database* and *payments*. Place a service in each.
3. Create an organization *engineering*, with two teams *backend* and *analytics*: navigate **Access Control** -> **Orgs & Teams** -> **Create** to create the organization, and **Access Control** -> **Orgs & Teams** -> *engineering* -> **+**, respectively.
4. Add Moby to the backend team, and create a user Abe in the analytics team.
5. Assign the *None* role from engineering to bigstore; assign full control to the backend team for the database and payments collections; and assign the *View Only* role to the analytics team for the database.
6. Optional: Draw the current RBAC configuration for the engineering organization and the bigstore collection on the board if possible.
7. Log in to UCP as Moby. Which services can Moby see, and why?
8. Log in to UCP as Abe. Which services can Abe see, and why?

2.3 Part 3: Kubernetes Namespaces & Role Bindings

1. As admin, create a Kubernetes namespace in UCP by entering the following in the yaml field under **Kubernetes** -> **Create**:

```
apiVersion: v1
kind: Namespace
metadata:
  name: myns
```

2. Create a pod in this namespace with the following yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
  namespace: myns
spec:
  containers:
```

```
- name: demo-nginx
  image: nginx
```

3. List your pods by navigating **Kubernetes -> Pods**; the `demo-nginx` pod is nowhere to be found. Navigate **Kubernetes -> Namespaces**, hover over `myns` and click **Set Context**; the `demo-nginx` pod will now appear in the Pods tab. UCP filters the Kubernetes resources it displays by namespace, unless you activate the *Set Context for all Namespaces* toggle on the Namespaces tab.
4. Finally, give the `edit` ClusterRole to the backend team from above for this namespace, via **Access Control -> Grants -> Kubernetes**.

2.4 Conclusion

In this demo, we clicked through basic setup of UCP users, teams, organizations, resource collections, namespaces, Swarm grants, and Kubernetes role bindings. Note the similarities between the two RBAC models; they are nearly identical, the possible nesting of Swarm resource collections notwithstanding.

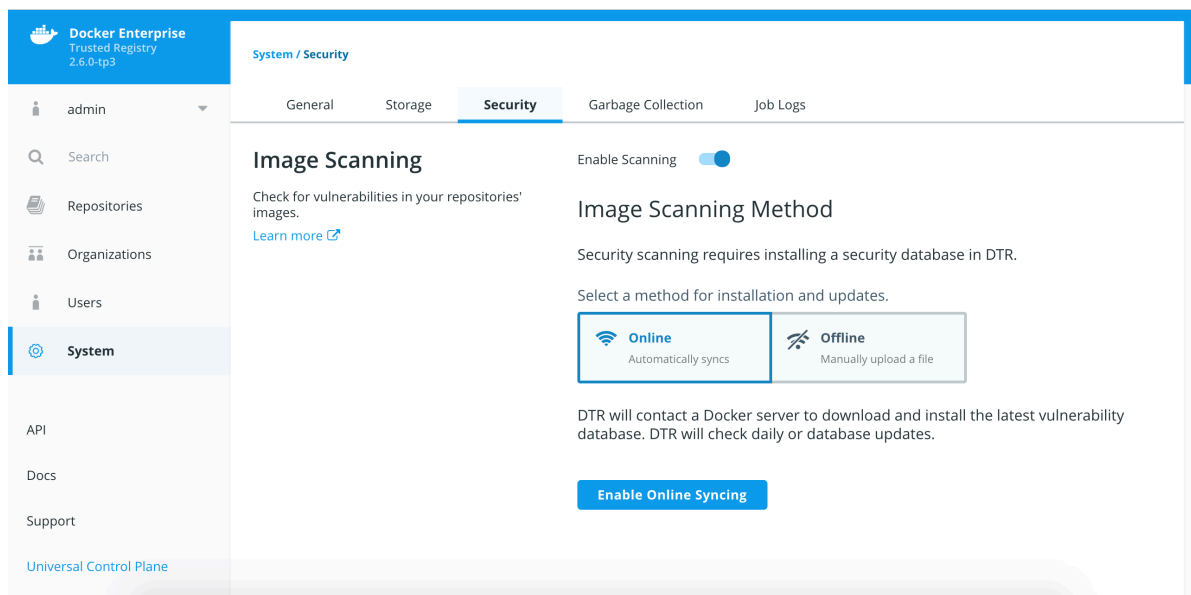
3 Instructor Demo: Security Scanning

In this demo, we'll illustrate:

- Setting up security scanning in DTR
- Configuring scans for an individual repository
- Interpreting scanning results in UCP and DTR

3.1 Setting Up Security Scanning

1. In DTR, click **System -> Security** and then toggle the **Enable Scanning** switch to turn on image security scanning. The two grayed out boxes will come into clear view and let you choose whether you'd like to enable automatic or manual scanning.



2. Click **Enable Online Syncing**. This will start the CVE database download.
3. Another section will pop up below that allows you to set the automatic scan timeout length. Set it to one hour.

Automatic Scanning Timeouts

Set a value to timeout scanning if it exceeds the provided value.

Scanning timeout limit

Hours

4. It'll take a few minutes to download your first CVE database; while that downloads, we can set up some repositories for scanning.

3.2 Configuring Scanning Options

1. Create a new repository in DTR named `admin/centos`.
2. Navigate **Repositories** -> **admin/centos** -> **Settings**, and then scroll down to the **Image Scanning** section and select **On push** to set it to scan automatically.

3.3 Scanning an Image

1. Download the `centos:7` image:

```
[centos@ucp-manager-0 ~]$ docker image pull centos:7
```

2. Set a variable and retag the image:

```
[centos@ucp-manager-0 ~]$ DTR_FQDN=<DTR FQDN>
[centos@ucp-manager-0 ~]$ docker image tag \
    centos:7 ${DTR_FQDN}:4443/admin/centos:sample
```

3. Navigate back to **System** -> **Security** in DTR and check the **Image Scanning** section to ensure that the CVE database has finished updating, as indicated by the *last sync* text above the **Sync Database now** button:

Image Scanning


Check for vulnerabilities in your repositories' images.
[Learn more](#)

Enable Scanning ☒


Image Scanning Method

Security scanning requires installing a security database in DTR.

Select a method for installation and updates.



Online
Automatically syncs



Offline
Manually upload a file

Last sync: Sep 30, 2018 @ 7:18 PM
 CVE Database version: 516

[Sync Database now](#)

4. Push the image:

```
[centos@ucp-manager-0 ~]$ docker image push ${DTR_FQDN}:4443/admin/centos:sample
```

5. Navigate to the image and click **View Details** to see that the security vulnerabilities are scanned automatically:

Repositories / admin / centos / sample / Layers

admin / centos : sample

linux / amd64 5182e96772bf 71.23 MB Pushed 21 seconds ago by admin

[Delete](#) [Promote](#) [Scan](#)

Layers Components

- 1 ADD file:6340c690b08865d7eb84a36050a0ab0e...
- 2 LABEL org.label-schema.schema-version=1.0 org.l...
- 3 CMD ["/bin/bash"]

Image scan
Scanning for this image is still in progress.

Notice Scanning for this image is still in progress.

6. Navigate back to your UCP dashboard and notice how memory and CPU usage are spiking from the scan.

3.4 Reading Scan Results

1. Back in DTR, navigate **Repositories -> admin/centos -> Tags -> View Details**. This will take you to the layer view of your image. You should see something like:

1	ADD file:6340c690b08865d7eb84a36050a0ab0e...	71.23 MB	
2	LABEL org.label-schema.schema-version=1.0 org.la...	Components (85)	Vulnerabilities
3	CMD ["/bin/bash"]		
		glibc@2.17-222.el7	2 Critical 3 Major
		zlib@1.2.8	2 Critical 2 Major
		bash@4.2.46-30.el7	2 Critical
		curl@7.29.0-46.el7	1 Critical 4 Major
		systemd@219-57.el7	1 Critical 1 Major
		cryptsetup@1.7.4-4.el7	1 Critical
		libidn@1.28-4.el7	1 Critical
		binutils@2.27-28.base.el7_5.1	5 Major
		kerberos@1.15.1-19.el7	5 Major
		openssl@1.0.2k-12.el7	3 Major
		libxml2@2.9.1-6.el7_2.3	3 Major

You'll see which and how many vulnerabilities are in each layer.

- Click on the **Components** tab on the same page. You'll see a view similar to this:

Layers	Components				
	<div><div>glibc</div><div><div>2.17-222.el7</div><div>2 Critical 3 Major</div></div></div>	<div><div>glibc</div><div>Version: 2.17-222.el7</div><div>License: LGPL</div></div>			
	<div><div>zlib</div><div><div>1.2.8</div><div>2 Critical 2 Major</div></div></div>	<div><div>Vulnerabilities</div><div>Severity</div><div>Description</div></div>			
	<div><div>bash</div><div><div>4.2.46-30.el7</div><div>2 Critical</div></div></div>	<div><div>CVE-2014-4043</div><div>7.5</div><div>The posix_spawn_file_actions_addopen function in glibc before 2.20 does not copy its path argument in accordance with the POSIX specification, which allows context-dependent attackers to trigger use-after-free vulnerabilities.</div></div>	<div><div>Show layers affected</div></div>		
	<div><div>curl</div><div><div>7.29.0-46.el7</div><div>1 Critical 4 Major</div></div></div>	<div><div>CVE-2016-2856</div><div>7.2</div><div>pt_chown in the glibc package before 2.19-18+deb8u4 on Debian jessie; the elibc package before 2.15-0ubuntu10.14 on Ubuntu 12.04 LTS and before 2.19-0ubuntu6.8 on Ubuntu 14.04 LTS; and the glibc package before 2.21-0ubuntu4.2 on Ubuntu 15.10 and before 2.23-0ubuntu1 on Ubuntu 16.04 LTS and 16.10 lacks a namespace check associated with file-descriptor passing, which allows local users to capture keystrokes and</div></div>	<div><div>Show layers affected</div></div>		
	<div><div>systemd</div></div>				

You will see the exact vulnerabilities, a short summary of each vulnerability, the severity rating of each, and be able to click a link to its entry in the official CVE database. Click on one of the CVE links for a vulnerability.

3.5 Overriding a Vulnerability

- Still in the *Components* view of the `admin/centos:sample`, click **Show layers affected** for any CVE of any component, and click **hide** to hide this vulnerability. At the top of the page displaying how many vulnerabilities there are of each severity level, there should now be an extra category showing *1 hidden*; this allows you to suppress notifications for vulnerabilities your security team have dismissed as nonthreatening.

3.6 Vulnerabilities in UCP

1. Start up a Swarm service on your cluster using your scanned image:

```
[centos@ucp-manager-0 ~]$ docker service create --name demo \
    ${DTR_FQDN}:4443/admin/centos:sample ping 8.8.8.8
```

Back on UCP, navigate **Swarm** -> **Services** to see the service, along with a summary of how many vulnerabilities the security scanner found in its underlying image.

3.7 Conclusion

In this demo, we explored image security scanning in DTR and delved deep into obtaining information about each vulnerability. For expedience, we set both database downloads and image scanning to happen automatically as soon as the necessary information (database updates and new image layers, respectively) was available. Since scanning can be resource intensive, it may be necessary to schedule scanning for low-usage times for your DTR. Either management strategy provides extra security transparency that can be added on to any container pipeline.